The Tcl9 Programming Language

A Comprehensive Guide

Second Edition

ASHOK P. NADKARNI

The Tcl Programming Language

A Comprehensive Guide

Second Edition

Ashok P. Nadkarni

The Tcl Programming Language: A Comprehensive Guide

Second Edition

Copyright © 2025 Ashok P. Nadkarni

All rights reserved. No part of this book may be reproduced, stored or transmitted by any means without the prior written permission of the author. If you purchased an electronic version of the book however, you may copy it to multiple devices owned by you. The author makes no warranty, expressed or implied, as to the accuracy of the book contents and assumes no liability for any damages arising from any inaccuracies or errors.

The `asciidoctor` and `asciidoctor-fopub` tools from Asciidoctor project were used to produce the content for this book. Print formatting was done with Apache FOP from the Apache Graphics Project. Text is typeset in the Noto Serif font from Google. Code samples use a combination of Noto Mono from Google and M+ 1p from the M+ Fonts Project.

To Aaee and Baba

For all the sacrifices

To Samyojita

How could I have been ever so lucky

To Devika and Akash

For the immeasurable joy you brought

Table of Contents

Preface xxv
1. Introduction 1
    1.1. A little bit of history 1
    1.2. What Tcl offers 1
    1.3. Reading this book 2
        1.3.1. Typographic conventions 2
        1.3.2. Utility procedures used in the book 4
    1.4. Online resources 4
2. Getting Started 5
    2.1. Installing Tcl 5
        2.1.1. Installing with system package managers 5
        2.1.2. Binary distributions 6
        2.1.3. Building from source 6
            2.1.3.1. Tcl source releases 6
            2.1.3.2. Build configurations 7
            2.1.3.3. Building on Unix-like platforms 7
            2.1.3.4. Building on Windows 8
            2.1.3.5. Building on macOS 8
        2.1.4. Reference documentation 9
    2.2. Running a Tcl program 9
        2.2.1. The Tcl library and interpreter 9
        2.2.2. The tclsh command-line shell 10
            2.2.2.1. Running tclsh interactively 10
            2.2.2.2. Running programs with tclsh 12
        2.2.3. The wish graphical shell 13
            2.2.3.1. Running wish interactively 13
            2.2.3.2. Running scripts with wish 14
        2.2.4. The tkcon enhanced shell 15
        2.2.5. Exiting a Tcl application 15
        2.2.6. Making Tcl scripts executable 15
            2.2.6.1. Executable scripts on Unix 15
            2.2.6.2. Executable scripts on Windows 16
    2.3. The application runtime environment 17
        2.3.1. Command-line arguments 17
        2.3.2. The working directory: pwd, cd 18
        2.3.3. Environment variables: env 18
        2.3.4. The process identifier: pid 19
        2.3.5. Executable file path: info nameofexecutable 19
        2.3.6. Tcl version information: info tclversion|patchlevel 19
        2.3.7. Tcl configuration: tcl::pkgconfig, tcl::build-info 20
        2.3.8. Platform information 21
3. Tcl Basics 23
    3.1. Basic syntax 23
    3.2. Substitutions 24
        3.2.1. Backslash substitutions 25

3.2.2. Variable substitutions 26
3.2.3. Command substitutions 27
3.3. Quoting 28
3.3.1. Quoting using double quotes 28
3.3.2. Quoting using braces 29
3.3.3. Choosing the quoting mechanism 30
3.4. Argument expansion 31
3.5. Commands 32
3.5.1. Command invocation 32
3.5.1.1. Unknown command handlers 33
3.5.2. Comments 34
3.5.3. Renaming a command 35
3.5.4. Deleting a command 36
3.5.5. Redefining commands 36
3.5.6. Enumerating commands: info commands 37
3.5.7. Command implementation types: info cmdtype 38
3.5.8. Command ensembles 39
3.5.9. Procedures 39
3.5.9.1. Defining procedures: proc 39
3.5.9.2. Procedure parameters 40
3.5.9.2.1. Default argument values 40
3.5.9.2.2. Variable number of arguments 41
3.5.9.2.3. Named parameters and options 41
3.5.9.3. Returning from a procedure: return 43
3.5.9.4. Anonymous procedures: apply 43
3.5.9.5. Introspecting procedures: info procs|args|default|body 45
3.6. Variables 46
3.6.1. Variable name syntax 46
3.6.2. Variable assignment: set 47
3.6.3. Getting a variable's value 47
3.6.4. Unsetting variables: unset 49
3.6.5. Variable scopes, lifetimes and visibility 49
3.6.5.1. Local variables 49
3.6.5.2. Global variables: global 50
3.6.5.3. Creation is not definition 50
3.6.6. Enumerating variables: info vars|locals|globals 51
3.6.7. Checking variable existence: info exists 52
3.6.8. Array variables 52
3.6.8.1. Basic array operations 52
3.6.8.2. Array defaults: array default 53
3.6.8.3. Checking for arrays: array exists 54
3.6.8.4. Checking for element existence: info exists, array names 54
3.6.8.5. Operating on multiple elements: array set|get|unset 54
3.6.8.6. Iterating over arrays: array for|startsearch|nextelement|anymore|donesearch 56
3.6.8.7. Array statistics: array size, array statistics 57
3.6.8.8. Printing an array: parray 58
3.6.8.9. More on array keys 58

3.6.9. Constant variables: const 59
3.6.10. Predefined variables 59
3.7. Conditional execution: if 60
3.8. Conditional execution: switch 61
3.9. Looping on a condition: while 63
3.10. Looping over values: for 64
3.11. Terminating loops: break 64
3.12. Skipping loops: continue 65
3.13. Evaluating strings: eval 65
3.13.1. Double substitutions in eval 66
3.14. Evaluating file content: source 68
3.14.1. Retrieving script paths: info script 68
3.15. Introspection 70
3.16. Getting error information 70
3.17. The EIAS principle 71
4. Strings 73
4.1. What is a string 73
4.1.1. Tcl and Unicode 73
4.2. String indices 74
4.3. String literals 75
4.4. Counting characters: string length 75
4.5. Retrieving a character by position: string index 75
4.6. Retrieving substring ranges: string range 75
4.7. Inserting characters: string insert 75
4.8. Appending characters: append 76
4.9. Replace or delete ranges: string replace 76
4.10. Replace or delete substrings: string map 77
4.11. Trimming character sets: string trim|trimleft|trimright 77
4.12. Concatenating strings: string cat 78
4.13. Joining strings with separators: join 79
4.14. Repeating strings: string repeat 79
4.15. Changing case: string tolower|toupper|totitle 79
4.16. Reversing a string: string reverse 80
4.17. Searching for substrings: string first|last 80
4.18. Searching for word boundaries 80
4.19. Customized interpolation: subst 81
4.20. Formatting strings: format 82
4.20.1. Conversion characters 83
4.20.2. XPG3 format position specifiers 84
4.20.3. Specifying minimum field widths 85
4.20.4. Format flags 85
4.20.5. Precision specifier 86
4.20.6. The size modifier 87
4.21. Parsing strings: scan 87
4.21.1. Conversion characters 89
4.21.2. Scan termination 90
4.21.3. XPG3 scan position specifier 91
4.21.4. Specifying maximum widths 92

4.21.5. The size modifier 92
4.22. Comparing strings: string equal|compare 93
4.23. String validation: string is 94
4.24. Glob pattern matching: string match 96
4.25. Matching shared prefixes: ::tcl::prefix 98
5. Lists 101
5.1. Basic list construction: list 101
5.2. List literals 101
5.3. List indices 104
5.3.1. Nested list indices 104
5.4. Retrieving elements by position: lindex 104
5.5. Extracting elements by position: lpop 105
5.6. Retrieving a sublist: lrange 105
5.7. Retrieving leading elements: lassign 105
5.8. Iterating over a list: foreach 106
5.9. Appending elements: lappend 107
5.10. Inserting elements: linsert 107
5.11. Setting element values: lset 108
5.12. Deleting elements: lremove 108
5.13. Replacing elements: lreplace, ledit 109
5.14. Counting elements: llength 110
5.15. Splitting strings into lists: split 110
5.16. Numeric sequences: lseq 111
5.17. Repeating elements: lrepeat 113
5.18. Concatenating lists: concat 113
5.19. Mapping list elements: lmap 113
5.20. Reversing a list: lreverse 114
5.21. Sorting lists: lsort 115
5.21.1. Comparing elements 115
5.21.2. Sort ordering 116
5.21.3. Sorting nested lists with -index 117
5.21.4. Sorting dictionaries with -stride 117
5.21.5. Retrieving sorted indices with -indices 118
5.21.6. Removing duplicate elements 119
5.22. Searching lists: lsearch 120
5.22.1. Search match operators 120
5.22.2. Search operand types 121
5.22.3. Searching nested lists 121
5.22.4. Searching grouped lists 122
5.22.5. Retrieving all matches 122
5.22.6. Retrieving element values 123
5.22.7. Searching sorted lists 123
5.22.8. Specifying a start offset 124
6. Dictionaries 125
6.1. Dictionary literals 125
6.2. Basic dictionary construction: dict create 126
6.3. Nested dictionaries 126
6.4. Dictionary and list compatibility 127

6.5. Checking for a key: dict exists ... 127
6.6. Retrieving the value for a key: dict get|getdef|getwithdefault ... 128
6.7. Enumerating dictionaries: dict keys|values ... 129
6.8. Setting values with dict set ... 129
6.9. Removing dictionary elements: dict unset|remove ... 130
6.10. Appending to string values: dict append ... 131
6.11. Appending list elements to values: dict lappend ... 131
6.12. Incrementing dictionary values: dict incr ... 132
6.13. Replacing multiple values: dict replace ... 132
6.14. Combining dictionaries: dict merge ... 132
6.15. Iterating over dictionaries: dict for ... 133
6.16. Mapping values: dict map ... 133
6.17. Filtering dictionaries: dict filter ... 134
6.18. Shadowing dictionaries with local variables: dict update ... 135
6.19. Shadowing nested dictionaries: dict with ... 136
6.20. Count of entries: dict size ... 138
6.21. Dictionary statistics: dict info ... 138
6.22. Dictionaries versus arrays ... 138
7. Numerics ... 141
7.1. Types and representations ... 141
7.1.1. The boolean type ... 141
7.1.2. The integer types ... 142
7.1.3. The floating point type ... 142
7.1.3.1. Floating point classification: fpclassify ... 143
7.1.4. Validation of types ... 143
7.1.5. Number conversions ... 144
7.1.5.1. Converting between strings and numbers ... 144
7.1.5.2. Converting between numeric types ... 144
7.2. Mathematical operations ... 145
7.2.1. The tcl::mathop commands ... 145
7.2.1.1. Arithmetic operator commands ... 146
7.2.1.2. Comparison operator commands ... 147
7.2.1.3. String operator commands ... 148
7.2.1.4. List operator commands ... 148
7.2.1.5. Bit-wise operator commands ... 149
7.2.2. Infix expressions: expr ... 149
7.2.2.1. Comments in expressions ... 150
7.2.2.2. Operands in expressions ... 151
7.2.2.3. Operators in expressions ... 151
7.2.2.4. Grouping operands with parenthesis ... 153
7.2.2.5. Braces and double substitution ... 153
7.2.3. Incrementing variables: incr ... 155
7.2.3.1. Expressions in other commands ... 155
7.3. Mathematical functions ... 156
7.3.1. Using functions in expressions ... 156
7.3.2. Defining custom functions ... 157
8. Binary data ... 159
8.1. Binary literals ... 159

8.2. Encoding binary strings as ASCII ... 160
8.2.1. Hexadecimal format: binary encode|decode hex ... 160
8.2.2. Base64 format: binary encode|decode base64 ... 160
8.2.3. Uuencode format: binary encode|decode uuencode ... 161
8.3. Constructing binary strings: binary format ... 161
8.3.1. Type specifiers for binary format ... 163
8.3.2. Cursor movement for formatting ... 166
8.4. Parsing binary strings: binary scan ... 166
8.4.1. Type specifiers for binary scan ... 168
8.4.2. Cursor movement for scanning ... 171
8.5. Compressing data ... 172
8.5.1. Compressing strings ... 172
8.5.1.1. Raw DEFLATE compression: zlib deflate|inflate ... 173
8.5.1.2. Zlib compression: zlib compress|decompress ... 173
8.5.1.3. Gzip compression: zlib gzip|gunzip ... 174
8.5.2. Compressing streams ... 175
8.5.2.1. Creating a zlib stream: zlib stream ... 175
8.5.2.2. Writing to a zlib stream ... 176
8.5.2.3. Finalizing a zlib stream: finalize, put -finalize ... 176
8.5.2.4. Reading from a zlib stream: get ... 176
8.5.2.5. Computing zlib stream checksum ... 177
8.5.2.6. Reusing a zlib stream: reuse ... 177
8.5.2.7. Closing a zlib stream: close ... 178
8.5.2.8. Decompression streams ... 178
8.5.2.9. Flushing zlib streams ... 178
8.6. Computing checksums: zlib adler32|crc32 ... 179
9. Globalization ... 181
9.1. Character encoding ... 181
9.1.1. Encoding profiles ... 182
9.1.2. Supported encodings: encoding names ... 183
9.1.3. Encoding characters: encoding convertto ... 183
9.1.4. Decoding characters: encoding convertfrom ... 184
9.1.5. Adding new encodings: encoding dirs ... 185
9.1.6. The system encoding: encoding system ... 185
9.1.7. Reading and writing encoded data ... 186
9.2. Internationalization ... 186
9.3. Localization ... 186
9.3.1. Locales ... 186
9.3.2. Message catalogs: mcset, mcmset, mcflset, mcflmset ... 187
9.3.3. Loading message catalogs: mcload, mcloadedlocales ... 188
9.3.4. Retrieving translations: mc ... 188
9.3.5. Comparing translation lengths ... 189
9.3.6. Retrieving and setting the locale: mclocale ... 189
9.3.7. Locale preferences: mcpreferences ... 190
9.3.8. Partitioning catalogs with namespaces: mcn ... 191
9.3.9. Unknown message keys: mcunknown, mcexists ... 192
9.3.10. Private package locales ... 193
9.3.10.1. Managing package locales: mcpackagelocale ... 193

9.3.10.2. Package locale options: mcpackageconfig 193
9.3.10.3. Package namespace: mcpackagenamespaceget 194
9.4. Internationalized Domain Names: tcl::idna 195
10. Regular Expressions 197
10.1. Matching regular expressions: regexp 198
10.1.1. Matching specific characters 198
10.1.2. Matching any character 198
10.1.3. Bracket expressions and character classes 199
10.1.4. Atoms and Groups 201
10.1.5. Quantifiers 202
10.1.6. Alternation and branches 203
10.1.7. Constraints 203
10.1.7.1. Anchoring with ^ and $ 203
10.1.7.2. Constraint escapes 204
10.1.7.3. Lookahead constraints 204
10.1.8. Back references 205
10.1.9. Counting number of matches 206
10.1.10. Retrieving matches 206
10.1.10.1. Retrieving matched content 206
10.1.10.2. Retrieving matched indices 207
10.1.10.3. Retrieving matches with -inline 207
10.1.10.4. Retrieving all matches 207
10.1.11. Option metasyntax 208
10.1.12. Case-independent matching 208
10.1.13. Matching literal strings 208
10.1.14. Newline-sensitive matching 209
10.1.15. Matching at an offset: -start 210
10.1.16. Controlling greediness 210
10.1.17. Comments and expanded syntax 211
10.2. Substituting regular expressions: regsub 212
10.2.1. Computed substitution with regsub 213
11. Dates and Time 215
11.1. POSIX seconds and the epoch 215
11.2. The Julian, Gregorian and alternate calendars 215
11.3. Time zones 216
11.4. Retrieving the current time: clock seconds | milliseconds | microseconds 217
11.5. Interval measurement: clock clicks 217
11.6. Formatting time for display: clock format 217
11.6.1. Formatting for a different time zone: -timezone, -gmt 218
11.6.2. Formatting for a locale: -locale 218
11.6.3. Controlling display format: -format 218
11.7. Parsing dates and times: clock scan 222
11.7.1. Specifying the parse format: -format 222
11.7.2. Specifying the time zone for parsing: -timezone, -gmt 223
11.7.3. Parsing localized time strings: -locale 223
11.7.4. Validating time strings: -validate 223
11.7.5. Changing the defaults for parsing: -base 224
11.7.6. Free form parsing of time strings 224

11.8. Time arithmetic: clock add 225
11.8.1. Clock computations 225
11.9. Localization 227
11.10. Time representation standards 227
12. Files and File Systems 229
12.1. File paths 229
12.1.1. Path syntax 229
12.1.2. Absolute and relative paths: file pathtype 230
12.1.3. Home directory and tilde substitution 230
12.1.4. Parsing paths: file dirname|extension|rootname|split|tail 231
12.1.5. Constructing paths: file join 232
12.1.6. Path normalization: file normalize 232
12.1.7. Converting paths to native form: file nativename 233
12.2. File properties and metadata 234
12.2.1. File size: file size 234
12.2.2. File timestamps: file atime|mtime 234
12.2.3. File information: file stat|lstat 234
12.2.4. Access checks: file exists|readable|writable|executable|owned 235
12.2.5. File types: file isdirectory|isfile|type 236
12.2.6. File attributes: file attributes 237
12.3. File system operations 239
12.3.1. File system information: file volumes|system|separator 239
12.3.2. Creating directories: file mkdir 239
12.3.3. Removing files and directories: file delete 240
12.3.4. Copying and renaming: file copy|rename 240
12.3.5. Enumerating files: glob 242
12.3.5.1. Matching based on type: -type option 243
12.3.5.2. Changing glob locations: -directory, -path 244
12.3.5.3. Stripping path names: -tails 245
12.3.5.4. Combining path component patterns: -join 245
12.3.5.5. Special considerations for glob 246
12.3.5.5.1. Case sensitivity 246
12.3.5.5.2. Short names on Windows 246
12.3.5.5.3. Enumerating hidden files 246
12.3.6. Links: file link, file readlink 247
12.3.7. Temporary files: file tempfile|tempdir 248
13. Channels and Basic I/O 249
13.1. Channels and File I/O 249
13.2. Standard channels: stdin, stdout, stderr 249
13.3. Creating file channels: open 250
13.4. Closing a channel: chan close, close 254
13.5. Channel configuration: chan configure, fconfigure 254
13.6. Writing to channels: chan puts, puts 255
13.6.1. Output buffering 255
13.6.1.1. Buffering mode: -buffering 255
13.6.1.2. Flushing buffers: chan flush, flush 256
13.6.1.3. Sizing buffers: -buffersize 256
13.7. Reading from channels 256

13.7.1. Reading lines from a file: chan gets, gets 257
13.7.2. Reading characters from a file: chan read, read 257
13.7.3. Detecting end of file: chan eof, eof 258
13.7.4. Input buffering 259
13.8. File utilities: writeFile, readFile, foreachLine 259
13.8.1. A utility to write files: writeFile 259
13.8.2. A utility to read files: readFile 259
13.8.3. Iterating over lines: foreachLine 259
13.9. Terminal configuration 260
13.9.1. Input character processing: -inputmode 260
13.9.2. Output screen size: -winsize 260
13.10. Newline translation: -translation 260
13.11. The end of file character: -eofchar 261
13.12. Channel encoding: -encoding 262
13.13. Encoding profiles: -profile 262
13.14. Binary I/O 263
13.15. The file access pointer 264
13.15.1. Retrieving the file access pointer: chan tell, tell 264
13.15.2. Setting the file access position: chan seek, seek 265
13.16. Truncating files: chan truncate 266
13.17. Copying data between channels: chan copy, fcopy 266
13.18. Enumerating open channels: chan names 267
14. Code Execution 269
14.1. Frames and the call stack 269
14.1.1. The call stack 269
14.1.2. Inspecting the call stack: info level 270
14.1.3. Commands that create call frames 272
14.1.4. Referencing variables in call frames: upvar 272
14.1.5. Executing scripts in a call frame: uplevel 276
14.1.6. The internal C stack 280
14.1.7. Recursing in place: tailcall 282
14.1.8. Hidden frames: info frame 286
14.2. Traces 288
14.2.1. Tracing variables: trace add variable 288
14.2.1.1. Tracing array variables 291
14.2.1.2. Applications of variable tracing 293
14.2.2. Tracing commands 296
14.2.2.1. Tracing command lifetimes: trace add command 296
14.2.2.2. Tracing command execution: trace add execution 297
14.2.3. Deleting a trace: trace remove 299
14.2.4. Inspecting traces: trace info 299
14.3. Code construction 300
14.3.1. Scripts versus command prefixes 300
14.3.1.1. Constructing command prefixes 301
14.3.1.2. Constructing scripts 302
14.3.2. Capturing namespace contexts in callbacks 302
14.4. Metaprogramming 302
14.4.1. Procedures with initializers 303

14.4.2. Parsing data 305
14.4.3. Code generalization 308
14.5. Command history: history 309
14.6. Counting command invocations: info cmdcount 312
15. Errors and Exceptions 313
15.1. Dealing with failures 313
15.2. Return codes and the option dictionary 314
15.2.1. Return codes 314
15.2.2. Return code propagation 316
15.2.2.1. Propagating **break** and **continue** return codes 316
15.2.2.2. Propagating the **return** return code 317
15.2.2.3. Propagating the **error** return code 319
15.2.3. The return options dictionary 319
15.3. The return command 319
15.3.1. Unwinding multiple levels of the call stack 321
15.3.2. Emulating other commands with return 324
15.3.3. Custom return codes 325
15.3.4. Custom return options dictionary 325
15.4. Trapping exceptions 326
15.4.1. Trapping exceptions: catch 326
15.4.2. The error stack and return options dictionary 327
15.4.2.1. Error stack trace: -errorinfo element, errorInfo 327
15.4.2.2. Error line number: -errorline element 327
15.4.2.3. Error codes: -errorcode element, errorCode 328
15.4.2.4. Error stack: -errorstack element, info errorstack 328
15.4.3. Trapping exceptions: try 329
15.5. Raising exceptions 332
15.5.1. Raising errors: throw, error 332
15.5.2. Raising errors: return -code 333
15.6. Forwarding exceptions 334
15.6.1. Forwarding exceptions with return 334
15.6.2. Forwarding exceptions with error 335
15.7. Custom control statements 336
16. Namespaces 339
16.1. Namespace basics 339
16.1.1. A simple namespace example 339
16.1.2. Namespace names and hierarchy 341
16.1.2.1. Inspecting namespace hierarchies: namespace current|parent|children 343
16.1.2.2. Manipulating names: namespace qualifiers|tail 343
16.1.3. Deleting a namespace: namespace delete 344
16.1.4. Checking namespace existence: namespace exists 344
16.2. Executing code in a namespace: namespace eval|inscope 345
16.2.1. Namespace contexts in callbacks: namespace code 346
16.3. Namespace variables: variable 347
16.4. Defining commands in a namespace 348
16.4.1. Namespace contexts in procedures 349
16.5. Name resolution 349

16.5.1. Resolving variable names 349
16.5.1.1. Variable resolution outside a procedure 349
16.5.1.2. Variable resolution in a procedure 350
16.5.1.3. Linking to namespace variables: namespace upvar 351
16.5.2. Resolving namespace names 351
16.5.3. Resolving command names 351
16.5.3.1. Importing names: namespace export|import|forget 352
16.5.3.2. Namespace paths: namespace path 354
16.5.3.3. Comparing namespace imports and paths 355
16.5.3.4. Handling unknown commands: namespace unknown 356
16.5.4. Introspecting name resolution: namespace which|origin 357
16.6. Namespace ensembles 359
16.6.1. Ensemble commands 359
16.6.2. Creating ensembles: namespace ensemble create 359
16.6.2.1. Naming an ensemble command 360
16.6.3. Configuring ensembles 361
16.6.3.1. Subcommand configuration: -subcommands, -map 361
16.6.3.2. Subcommand prefixes: option -prefixes 362
16.6.3.3. Subcommand positioning: option -parameters 363
16.6.4. Handling unknown subcommands: option -unknown 364
16.6.5. Checking for ensembles: namespace ensemble exists 366
16.6.6. Nested ensembles 366
16.6.7. Examples of ensembles 367
17. Libraries and Packages 373
17.1. The Tcl system library 373
17.2. Loading libraries on demand: auto_load 374
17.2.1. The tclIndex files: auto_mkindex 374
17.3. Packages 375
17.3.1. Naming packages 375
17.3.2. Package versioning 375
17.3.2.1. Package version syntax 375
17.3.2.2. Comparing package versions: package vcompare|vsatisfies 376
17.3.3. Introspecting packages: package names|version|files 377
17.3.4. Installing packages 378
17.3.5. Searching for libraries 378
17.3.6. Loading packages: package require 378
17.3.6.1. Choosing stable versus unstable packages: package prefer 379
17.3.7. Checking if a package is loaded: package present 380
17.3.8. Registering packages: package ifneeded 380
17.3.9. Creating packages: package provide 381
17.4. Shared library extensions 382
17.4.1. Loading extensions: load 383
17.4.2. Enumerating loaded extensions: info loaded 384
17.5. Modules 384
17.5.1. Module file names 385
17.5.2. Searching for modules 385
17.5.3. Installing modules 387
17.5.4. Creating modules 387

17.6. Packages versus modules ... 388
17.7. Multiplatform packaging: platform package ... 388
17.7.1. The platform::shell package ... 389
17.8. Introspecting package configuration ... 390
18. Object-Oriented Programming ... 391
18.1. Objects and classes ... 391
18.2. Class basics ... 392
18.2.1. Creating a class ... 392
18.2.2. Class definition script ... 393
18.2.3. Destroying classes ... 394
18.2.4. Data members ... 394
18.2.4.1. Instance variables: variable, my variable ... 394
18.2.4.2. Class variables: classvariable ... 395
18.2.5. Methods ... 396
18.2.5.1. Constructors and destructors ... 396
18.2.5.2. Defining methods: method ... 397
18.2.5.3. Method visibility ... 397
18.2.5.4. The unknown method ... 398
18.2.5.5. Class methods: classmethod, myclass ... 399
18.2.5.6. Deleting methods: deletemethod ... 400
18.2.5.7. Renaming methods: renamemethod ... 400
18.2.5.8. Method callbacks: callback, mymethod ... 401
18.2.5.9. Methods as commands ... 402
18.2.6. Slot operations ... 402
18.2.7. Modifying an existing class ... 403
18.2.8. Class initializer: initialize ... 403
18.3. Working with objects ... 403
18.3.1. Creating an object: OBJECT create|new ... 403
18.3.2. Destroying objects ... 404
18.3.3. Invoking methods ... 404
18.3.4. Namespace contexts ... 405
18.3.5. External access to data members: my varname ... 405
18.4. Inheritance: superclass ... 406
18.4.1. Methods in derived classes ... 407
18.4.1.1. Chaining methods ... 407
18.4.2. Data members in derived classes ... 408
18.4.3. Multiple inheritance ... 409
18.4.4. Private contexts ... 410
18.4.4.1. Defining private contexts: private ... 411
18.4.4.2. Private methods and forwards ... 412
18.4.4.3. Private variables ... 413
18.5. Specializing objects: oo::objdefine ... 414
18.5.1. Object definition script ... 414
18.5.2. Object-specific methods ... 415
18.5.3. Changing an object's class ... 417
18.6. Using mix-ins ... 418
18.6.1. Adding a mix-in to a class: mixin ... 418
18.6.2. Using multiple mix-ins ... 419

18.6.3. Mix-ins versus inheritance ... 420
18.7. Method forwarding ... 420
18.8. Filter methods ... 422
18.8.1. Defining a filter class ... 423
18.8.2. When to use filters ... 424
18.9. Method chains ... 424
18.9.1. Method chain order ... 425
18.9.2. Method chain for unknown methods ... 426
18.9.3. Retrieving the method chain for a class ... 426
18.9.4. Inspecting method chains within method contexts ... 427
18.9.5. Looking up the next method in a chain ... 427
18.9.6. Controlling invocation order of methods ... 429
18.10. Programming without classes ... 430
18.11. Metaclasses ... 431
18.11.1. Implementing a metaclass ... 433
18.11.2. Abstract classes: oo::abstract ... 434
18.11.3. Singleton classes: oo::singleton ... 435
18.11.4. Configurable properties: oo::configurable ... 436
18.12. OO introspection ... 438
18.12.1. Enumerating objects and classes ... 438
18.12.2. Checking if an object is a class ... 439
18.12.3. Inspecting class relationships ... 439
18.12.4. Retrieving class definition namespaces ... 440
18.12.5. Object identity ... 440
18.12.6. Inspecting an object's class membership ... 441
18.12.7. Checking if a command is an object ... 442
18.12.8. Enumerating methods ... 443
18.12.9. Retrieving method definitions ... 444
18.12.10. Inspecting method chains and contexts ... 445
18.12.11. Inspecting filters ... 445
18.12.12. Enumerating variables ... 447
18.12.13. Enumerating configurable properties ... 447
19. The Event Loop ... 449
19.1. Event sources and types ... 449
19.2. The Tcl event loop ... 450
19.2.1. The event and idle task queues ... 450
19.2.2. Event loop operation ... 450
19.2.3. Running the event loop ... 451
19.2.3.1. Processing events based on conditions: vwait ... 451
19.2.3.1.1. Avoiding deadlocks with vwait ... 453
19.2.3.2. Single invocation: update ... 454
19.2.4. Event handlers and the call stack ... 455
19.3. Scheduling execution of code: after ... 456
19.3.1. Suspending execution ... 456
19.3.2. Scheduling code ... 456
19.3.3. Running on idle: after idle ... 457
19.3.3.1. Avoiding event queue starvation ... 458
19.3.4. Cancelling tasks: after cancel ... 460

19.3.5. Introspecting after handlers: after info 461
19.4. Event loop error handling 461
19.4.1. Custom background error handling: interp bgerror 462
20. Processes and Pipelines 463
20.1. Executing child processes: exec 463
20.1.1. Passing program arguments 464
20.1.2. Locating programs 465
20.1.2.1. Locating shell internal commands: auto_execok 465
20.1.3. Redirecting I/O 466
20.1.3.1. Redirecting input 466
20.1.3.2. Redirecting output 468
20.1.4. Error handling in exec 471
20.1.5. Running background processes 474
20.1.6. Limitations in exec 474
20.2. Channels for process pipelines: open 474
20.2.1. Running tclsh in a pipeline 476
20.2.2. Pipeline process ids: pid 479
20.2.3. Error handling in pipelines 479
20.3. Standalone pipes: chan pipe 479
20.4. Half-closing of channels 483
20.5. Passing environment to child processes 483
20.6. Managing child processes: tcl::process 484
20.6.1. Enumerating subprocesses: tcl::process list 484
20.6.2. Checking child process status: tcl::process status 484
20.6.3. Cleaning up process resources: tcl::process purge|autopurge 485
21. Advanced I/O 487
21.1. Asynchronous I/O 487
21.1.1. Non-blocking I/O 488
21.1.1.1. Changing the blocking mode for a channel 488
21.1.1.2. Checking if a channel is blocked 488
21.1.1.3. Non-blocking input 488
21.1.1.3.1. Reading lines in non-blocking mode: chan gets, gets 488
21.1.1.3.2. Reading characters in non-blocking mode: chan read, read 491
21.1.1.4. Non-blocking output: chan puts, puts 492
21.1.2. Event driven I/O: chan event, fileevent 493
21.1.3. Closing non-blocking channels 496
21.1.4. An interactive command line 496
21.2. Channel transforms 498
21.2.1. Channel transform basic operation 498
21.2.2. Implementing channel transforms 499
21.2.2.1. Initializing channel transforms 501
21.2.2.2. Finalizing channel transforms 501
21.2.2.3. Transforming data 502
21.2.2.4. Buffering in channel transforms 503
21.2.2.5. Limiting read-ahead 504
21.2.3. Using channel transforms: chan push|pop 504
21.2.4. Zlib channel transforms 505

21.3. Reflected channels ... 507
21.3.1. Implementing reflected channels ... 507
21.3.1.1. Initializing a reflected channel ... 508
21.3.1.2. Closing a reflected channel ... 509
21.3.1.3. Configuring a reflected channel ... 509
21.3.1.4. Non-blocking mode and event driven I/O ... 510
21.3.1.5. Implementing data output ... 512
21.3.1.6. Implementing data input ... 514
21.3.1.7. Reflected channel creation: chan create ... 514
21.3.1.8. Seeking in a reflected channel ... 515
21.3.1.9. The complete channel implementation ... 516
21.3.2. Using reflected channels ... 518
21.3.3. Reflected channel limitations ... 519
22. Networking and Communications ... 521
22.1. Network communications ... 521
22.1.1. Writing TCP clients: socket ... 521
22.1.1.1. Connecting synchronously ... 522
22.1.1.2. Connecting asynchronously ... 522
22.1.2. Writing TCP servers: socket -server ... 524
22.1.3. Socket configuration ... 526
22.1.3.1. Text and binary protocols ... 527
22.1.4. The http package ... 527
22.2. Communication over serial ports ... 529
22.2.1. Serial port buffer and queue sizes ... 529
22.2.2. Serial port speed, parity, and bit lengths ... 529
22.2.3. Serial port flow control ... 529
22.2.4. Timers related to serial ports ... 530
22.2.5. Checking for serial port errors ... 530
23. Interpreters ... 531
23.1. Creating interpreters: interp create ... 531
23.2. Identifying interpreters ... 532
23.3. Inspecting the interpreter hierarchy: interp children|exists ... 533
23.4. Destroying interpreters ... 533
23.5. Evaluating scripts in an interpreter: interp eval ... 533
23.6. Command aliases ... 534
23.6.1. Defining aliases: interp alias ... 534
23.6.2. Deleting aliases ... 536
23.6.3. Introspecting aliases: interp aliases|target ... 536
23.7. Execution context in child interpreters ... 537
23.8. Cancelling script evaluation: interp cancel ... 537
23.9. Sharing channels in interpreters ... 539
23.10. Safe interpreters ... 540
23.10.1. Creating a safe interpreter ... 540
23.10.2. Aliasing in safe interpreters ... 541
23.10.2.1. Precautions for aliased commands ... 542
23.10.3. Hidden commands ... 542
23.10.3.1. Invoking hidden commands: interp invokehidden ... 542
23.10.3.2. Hiding and exposing commands: interp hide|expose ... 544

23.10.3.3. Introspecting hidden commands: interp hidden 545
23.10.4. Trusting safe interpreters: interp marktrusted 545
23.10.5. Safe Tcl 546
23.10.5.1. Creating a Safe Tcl interpreter: safe::interpCreate|interpInit 547
23.10.5.2. Deleting a Safe Tcl interpreter: safe::interpDelete 547
23.10.5.3. Configuring a Safe Tcl interpreter: safe::interpConfigure 548
23.10.5.4. Safe Tcl file paths 548
23.10.5.5. Safe Tcl package search 549
23.10.5.6. Troubleshooting Safe Tcl interpreters 549
23.11. Setting resource limits 550
23.11.1. The recursion limit: interp recursionlimit 550
23.11.2. Limiting number of commands: interp limit 550
23.11.3. Limiting interpreter duration: interp limit 552
23.12. Examples using multiple interpreters 553
23.12.1. A safe network server 553
23.12.2. Implementing domain specific languages 555
24. Coroutines 563
24.1. Creating coroutines: coroutine 564
24.2. Suspending and resuming coroutines 564
24.2.1. Yielding to the caller: yield 565
24.2.2. Yielding after initialization 567
24.2.3. Yielding to arbitrary commands: yieldto 568
24.3. Checking coroutine context: info coroutine 569
24.4. Coroutine termination 570
24.4.1. Releasing resources on termination 571
24.5. Exception handling in coroutines 572
24.6. Variable scopes in coroutines 573
24.6.1. Private variables 574
24.7. Coroutines, uplevel and upvar 575
24.8. Coroutines and multiple interpreters 575
24.9. Code injection: coroprobe, coroinject 576
24.10. Using coroutines 578
24.10.1. Explicit and implicit state 578
24.10.2. Generators 579
24.10.3. Emulating objects 582
24.10.4. Producers, consumers, transformers and filters 583
24.10.5. Coroutines and the event loop 586
24.10.6. Emulating blocking calls: coroutine::util 587
24.10.7. Co-operative multitasking 588
25. ZipFS and Single File Deployment 597
25.1. Creating ZIP archives 597
25.1.1. Zipping directories: zipfs mkzip 598
25.1.2. Zipping list of files: zipfs lmkzip 599
25.2. The ZipFS file system 599
25.2.1. The ZipFS root 599
25.2.2. Mounting ZIP archives as a VFS: zipfs mount 599
25.2.3. Mounting in-memory ZIP : zipfs mountdata 600
25.2.4. Introspecting ZipFS mounts: zipfs mount 600

25.2.5. Dismounting ZipFS file systems: zipfs unmount 601
25.2.6. ZipFS utilities: zipfs exists|info|list|find|canonical 601
25.2.7. Accessing ZipFS files 602
25.3. Single file deployment 603
25.3.1. Embedded ZIP archives 603
25.3.2. Single-file applications 604
26. Testing and Performance 607
26.1. Testing 607
26.1.1. The tcltest package 607
26.2. Improving performance 611
26.2.1. Profiling scripts 611
26.2.2. Timing scripts 612
26.2.3. The timerate command 612
26.2.4. The time command 614
A. Core packages 615
A.1. The Tk graphical toolkit 615
A.2. The tdbc extension 615
A.3. The http and cookiejar packages 615
A.4. The Thread extension 615
A.5. The tclvfs extension 615
A.6. The registry extension 615
A.7. The dde extension 615
A.8. The IncrTcl extension 616
B. Utility scripts 617
Index 619

List of Figures

2.1. The wish windowing shell 13
2.2. A sample Tk window 14
14.1. Initial call frame 269
14.2. Level 1 call frame 270
14.3. Call stack and upvar 274
14.4. Call stack and uplevel 277
14.5. C stack and call frames 281
14.6. Call stack with tailcall 283
19.1. Call stack in an event handler 455
21.1. Basic channel operation 499

List of Tables

2.1. Configure options 7
2.2. Command-line argument globals 17
2.3. Platform information 21
3.1. Backslash sequences 25
3.2. Command types 38
3.3. Matching options for switch 62
4.1. Integer specifiers for format 83
4.2. String specifiers for format 83
4.3. Floating point specifiers for format 84
4.4. Flag component in format specifiers 86
4.5. Size modifiers for format 87
4.6. Integer specifiers for scan 89
4.7. String specifiers for scan 89
4.8. Integer size modifiers for scan 93
4.9. String validation classes 95
4.10. Pattern matching characters 97
5.1. Lsort comparison options 115
5.2. Lsearch matching options 120
5.3. Lsearch data type options 121
6.1. Differences between tables and arrays 139
7.1. Floating point classes 143
7.2. Arithmetic operators 146
7.3. Comparison operators 147
7.4. String comparison operators 148
7.5. List membership operators 148
7.6. Bit operators 149
7.7. Expression operators in precedence order 152
7.8. Mathematical functions 156
8.1. Type specifiers for binary format 163
8.2. Binary format cursor movement characters 166
8.3. Type specifiers for binary scan 168
8.4. Binary scan cursor movement characters 171
8.5. Gzip header keys 174
8.6. Zlib stream options 175
8.7. Zlib stream put options 176
9.1. Package locale options 194
10.1. Basic regular expression syntax 197
10.2. Regular expression character classes 200
10.3. Character class shorthands 201
10.4. Regular expression quantifiers 202
10.5. Constraint escape sequences 204
11.1. Format groups for clock 219
12.1. File stat array elements 235
12.2. Unix file attributes 237
12.3. Windows file attributes 238

12.4. macOS file attributes 238
12.5. Glob patterns 242
12.6. Glob category 1 type specifiers 243
12.7. Glob category 2 type specifiers 244
13.1. Access modes for open - string form 251
13.2. Access modes for open - list form 251
13.3. Buffering policy option values 256
13.4. Option -inputmode values 260
13.5. Option -translation values 261
13.6. Origin values for seek 265
14.1. Trace operations on variables 289
14.2. Trace operations on commands 296
14.3. Trace operations on command execution 297
15.1. Tcl-defined return codes 315
17.1. Package version requirements syntax 376
18.1. Class definition commands 393
18.2. TclOO slot operations 402
18.3. Object definition commands 415
18.4. oo::configurable property command options 437
19.1. Options controlling vwait conditions 452
19.2. Options for vwait event types 452
20.1. Access mode for pipelines using open 475
21.1. Channel transform subcommands 500
21.2. zlib push command options 506
21.3. Reflected channel subcommands 507
22.1. Socket-specific configuration options 526
22.2. Values for handshake configuration 530
23.1. Safe Tcl predefined aliases 546
23.2. Safe Tcl configuration options 548

Preface

...or why I wrote this book.

This book, as the more perceptive readers would have deduced from the title, is about the Tcl programming language.

About Tcl

Tcl was one of the first "dynamic" languages to become popular, seeing widespread use beginning with the early 90's. Along with its accompanying graphical programming toolkit Tk, the language was influential enough for its inventor John Ousterhout to be conferred the ACM Software System Award in 1998. Since that time, Tcl has found its way into every application category you can imagine and Tcl deployments run the gamut from embedded devices to distributed back-end infrastructure.

Yet, despite its wide use and adoption, Tcl has not gained the notoriety of the newer languages that have sprung up in the past few years. In large part this is because the Tcl community as a whole has never been particularly interested in evangelizing the language. This book hopes to remedy that by providing comprehensive coverage of Tcl 9, starting with the basics and continuing to the advanced facilities that distinguish the language.

About the book

I have attempted to cover every single feature of language. However, space limitations do not permit inclusion of the myriad libraries and extensions to Tcl. Even the graphical toolkit Tk, which is associated so strongly with Tcl that most people refer to the two in conjunction as Tcl/Tk, is not included. It would double the size of the book by itself. For the same reason, the book also does not discuss the C programming interface to Tcl although ease of interfacing to C is one of the strengths of the language.

About me

> Give me a place to stand and I'll move the earth.
>
> — Archimedes *famous Greek mathematician*

> Give me Tcl and I'll ship on time.
>
> — Me *famous Tcl author*

Authoring is hard work. Now a fainéant I anient[1] so I don't mind hard work. But it is **tedious** hard work (Oh Lord, indexing!). And there is not much money to be made in technical books, it being difficult to insert violence and gratuitous sex into the prose to sell a few more copies. So what was my motivation in writing this book? Likely the same as most authors of technical books — when you believe in a language or technology, you have an urge, a compulsion, to spread the word.

My association with Tcl includes intermittent stretches over decades. I have run engineering in start-ups where significant components of the product were developed in Tcl. I have also

[1] With apologies to Ogden Nash

authored several open source extensions and libraries, contributed to the feature set for Tcl 9 and am now a member of the Tcl Core Team. In that time I grew to be increasingly enamoured of the language for one primary reason — productivity. From the perspective of an individual programmer, this productivity results from Tcl's malleability and rich feature set that facilitate a number of different programming styles and architectural patterns. From a management perspective, Tcl is stable, portable and versatile enough to be of benefit throughout a product's life-cycle — from development, test, deployment to field support.

My hope is that this book will in some small way help popularize Tcl — beat the drum, so to speak, with regard to its simplicity and power.

About the Second Edition

The second edition of the book targets version 9.0 of the Tcl language. Differences with respect to Tcl 8.6 are highlighted where applicable.

Due to page limits in the print edition, the addition of new material has necessitated removal of some topics present in the first edition that are not part of the core language. The deleted material is available from the book website[2].

Acknowledgements

First up is of course Professor John Ousterhout without whom there would be no Tcl. Beyond that, I thank en masse the members of the Tcl Core Team (TCT) that now directs development and future direction of the language, the contributors to the Tcler's Wiki[3], which contains a trove of illustrative examples and expository material about Tcl, the Tcl Chat room where much illuminating discussion takes place, and the many individuals who are collectively responsible for the libraries and tools that complete the Tcl ecosystem. Special thanks to Arjen Markus and Doran Moppert who reviewed portions of the first edition.

Last but not least, a shout out to the Asciidoctor[4] team whose open source publishing toolchain greatly eased the actual process of writing and producing both electronic and print formats. Outstanding software.

Contacting me

You may contact me via my website https://www.magicsplat.com which also happens to be the home of my Tcl-related blog, articles and software. Alternatively, you may reach me through the Tcler's Wiki or the comp.lang.tcl newsgroup.

Ashok P. Nadkarni
Bengaluru, February 2025

[2] https://www.magicsplat.com/ttpl/index.html
[3] https://wiki.tcl-lang.org
[4] https://asciidoctor.org

1

Introduction

1.1. A little bit of history

The first version of Tcl sprang to life at UC Berkeley way back in 1988. Professor John Ousterhout's primary motivation[1] was to create a standardised, extensible language that could be easily embedded into applications to allow their functionality to be scripted. The accompanying graphical toolkit Tk, which used Tcl as its scripting language, came into being a couple of years later. The combination grew in popularity and was influential enough for Prof. Ousterhout to receive the ACM Software System and USENIX STUG awards in 1998.

Since those early years, Tcl has grown from an "embeddable, scripting" language to a full fledged dynamic programming language versatile enough for one-line throwaways to end-user facing applications to server backends.

1.2. What Tcl offers

The benefits of software development in Tcl stem from

- portability across all mainstream operating systems and even embedded environments.
- an interactive mode (Section 2.2.2.1) that encourages rapid experimentation, iterative prototyping and test-driven development.
- a consistent syntax, enabling advanced capabilities like metaprogramming (Section 14.4), custom control structures (Section 15.7), and domain specific languages (Section 23.12.2).
- an advanced file and I/O framework supporting custom I/O channels (Section 21.3) with data transforms (Section 21.2), and virtual file systems (Section A.5).
- an integrated event loop (Chapter 19), coroutines (Chapter 24) and threads (Section A.4) that provide the foundation for multiple concurrency models.
- object-oriented constructs (Chapter 18) that support class-based, object-based or prototype-based paradigms.
- programming conveniences like support for the full Unicode range, infinite precision arithmetic (Section 7.1.2) and traces for reactive programming (Section 14.2).
- multiple isolated execution environments (Chapter 23) with sandboxing (Section 23.10) for executing untrusted code.
- optional single file executable packaging (Section 25.3) for ease of deployment.

[1] https://www.tcl-lang.org/about/history.html

No doubt by now you are chomping at the bit to get started on Tcl. However, tradition demands we say a little bit about the book itself first.

1.3. Reading this book

For the newcomers...

The book requires no prior experience with Tcl but does assume some basic programming background on the reader's part. Advanced constructs like asynchronous programming, threads and coroutines require a little more sophistication but you can get a lot of programming done without venturing into these areas. My suggestion for newcomers would be to not get bogged down by the minutiae of every command. You can come back to refer to the details as and when needed. Experimentation in Tcl's interactive mode is the key to developing proficiency.

For the old hands...

For readers who have worked with Tcl before, the book serves as a reference with a detailed table of contents and a comprehensive index. At the same time, browsing through the book may very well lead many to discover new Tcl features and capabilities they might not have been aware of, or to gain a deeper understanding of advanced topics.

1.3.1. Typographic conventions

Below is the obligatory section on formatting and typographic conventions that are obvious to everyone but the publisher.

Text formatting

Within the text, we use *italics* to define terms and **bold** for emphasis. File paths and program elements like commands and variables are shown in a `monospace` font. Additionally, we use capitalized italics in the same monospace font for *`PLACEHOLDERS`* that stand for some variable part in a code fragment.

Code samples

Code samples fall into three categories:

- syntax descriptions
- commands typed at the Tcl shell prompt
- scripts as they might be stored in files

All use the same font employed for code within descriptive text.

The first of these are intended to show syntax of commands and not expected to be executed as-is. Optional parts of the command are shown enclosed in `?` characters.

```
set VARNAME ?VALUE?
```

The above syntax indicates that *`VARNAME`* and *`VALUE`* are only placeholders for the actual variable name and value, and that *`VALUE`* is optional.

Commands that you might type at the Tcl shell prompt are shown as

```
% set x 1
→ 1
```

The `%` character is the Tcl shell prompt so the command itself is `set x 1`. Any output that the shell prints out is prefixed with the `→` character. Depending on the example, lines may be truncated, indicated by ellipsis `…` , or wrapped, prefixed with a `↳` character.

Error messages printed by the shell are prefixed with `Ø`.

```
% set x $nosuchvariable
Ø can't read "nosuchvariable": no such variable
```

In the interest of saving space, short commands and the result may be shown on the same line without the prompt.

```
format %x 42 → 2a
format %b 42 → 101010
```

In this case, a result that is an empty string is shown as in the example below.

```
set x "" → (empty)
```

Finally scripts where the output of individual commands is not important or relevant are shown without the command prompt. Only the output of the last command is shown.

```
proc add {a b} {
    return [expr {$a+$b}] ❶
}
add 2 3
→ 5
```

❶ `expr` computes arithmetic expressions

The numbered callout shown in the above example is intended to either highlight or provide additional information about a line in the script.

Sidebars

Material that is related to the discussion but not directly relevant is placed in a sidebar. For example,

History of driving regulations

Licenses were not required for driving in the United States until 1903 when Massachusetts and Missouri became the first states to make them mandatory.

Highlighting

Certain notes and points of emphasis are highlighted in one of the following ways:

Important You must carry your driver's license and insurance papers at all times. *Stresses important points you must keep in mind.*

Caution Dangerous curves ahead. Reduce speed. *Actions to be carried out with some care.*

Warning Do not drink and drive. *Actions you must not do.*

Note Turning right on red is not permitted in New York. *Relevant information worth noting.*

Tip Use the exact change lanes for quicker service. *Tips for productivity.*

1.3.2. Utility procedures used in the book

Throughout the book we use various simple utility procedures for convenience, for example to print a list. These procedures are shown in Appendix B.

1.4. Online resources

The book's website is at https://www.magicsplat.com/ttpl. Chapters from the first edition that are absent in the second are available here as is the errata for the book.

The primary website for Tcl itself is https://www.tcl-lang.org.

The Tcler's Wiki[2] is where you should go for all kinds of tips, code samples, and wide ranging discussions on a variety of Tcl-related topics.

The Usenet group `comp.lang.tcl` is dedicated to a discussion of Tcl-related topics and a good place to get any questions answered. An alternative for interactive discussions is the Tcl Chatroom[3] which is accessible either via Slack, XMPP clients or through IRC gateways.

The Tcl source code repository is hosted at https://core.tcl-lang.org/ under Fossil[4] version control. This also hosts Tcl's ticketing[5] system. Official releases are available from the SourceForge download[6] area. Binaries are available from multiple sources (Section 2.1).

[2] https://wiki.tcl-lang.org
[3] https://wiki.tcl-lang.org/page/Tcl+Chatroom
[4] https://www.fossil-scm.org
[5] https://core.tcl-lang.org/tcl/ticket
[6] https://sourceforge.net/projects/tcl/files/Tcl/

2

Getting Started

We will begin by describing the procedure for installing Tcl and then move on to the actual mechanics of running Tcl programs.

2.1. Installing Tcl

There are several options for installing Tcl on your system:

- Via the operating system package manager
- Precompiled binaries from third parties
- Custom builds from build servers
- Building from source

2.1.1. Installing with system package managers

Package managers may not always have the latest Tcl version in their repository. At the time of writing, most have not been updated to Tcl 9 which has only very recently been released.

Linux

On Linux, Tcl can be installed with the system package manager, for example using `apt-get` on a Debian system and `yum` on Fedora.

```
apt-get install tcl
yum install tcl
```

Tcl extensions are generally available as separate packages.

Windows

Tcl can be installed with either of winget[1] and Chocolatey[2] package managers if they are installed on the system.

```
winget install Magicsplat.TclTk
choco install magicsplat-tcl-tk
```

[1] https://learn.microsoft.com/en-us/windows/package-manager/
[2] https://chocolatey.org/

Both `winget` and `choco` re-package the author's Magicsplat distribution. You may prefer to directly install using the original Magicsplat installer[3] to ensure you have the latest release.

MacOS

Although MacOS provides a system-installed Tcl, at the time of writing it was hopelessly out of date. Newer versions can be installed with either Homebrew or MacPorts. Even then, the caveats about outdated releases apply.

```
brew install tcl-tk
port install tcl
```

2.1.2. Binary distributions

There are several binary distributions of Tcl, some of which include thirdparty packages as well. These are listed on the Tcl download page[4]. Before downloading, check that the Tcl version and licensing restrictions meet your requirements.

2.1.3. Building from source

Finally, there is always the option to build Tcl directly from the official source releases.

Although Tcl itself is straightforward to build, it can be more involved to build third party extensions due to dependencies, differing build systems etc. The **Build Automation with Tcl** system (BAWT[5]) specifically tackles this problem. It supports Windows, Linux and macOS and requires the user to only run a single batch or shell script to build Tcl and the extensions of interest.

2.1.3.1. Tcl source releases

The released source archives are available from the Tcl distributions[6] page or directly from the SourceForge download area[7]. Download the following archives from the appropriate release-specific directory:

- tcl*VERSION*-src.tar.gz or tcl*VERSION*-src.zip which contain the source code for Tcl and some core packages. The two only differ in the archive format.
- tk*VERSION*-src.tar.gz or tk*VERSION*-src.zip which contain the source for the Tk extension. This is strictly not part of Tcl itself but you will need it if you want to use the GUI version of the Tcl/Tk shell (`wish`).

The steps outlined here describe building Tcl 9. Previous versions have some differences with respect to supported options.

[3] https://www.magicsplat.com/tcl-installer/index.html
[4] https://www.tcl-lang.org/software/tcltk/bindist.html
[5] https://www.bawt.tcl3d.org/documentation.html
[6] https://www.tcl-lang.org/software/tcltk/download.html
[7] https://sourceforge.net/projects/tcl/files/Tcl/

2.1.3.2. Build configurations

The core components of all Tcl applications are

- The main executable that is run by the user — a standard Tcl shell or a custom program
- The Tcl interpreter which implements the Tcl language
- Support Tcl scripts consisting of initialization code, message catalogs etc.

The location of these components on the file system depends on the configuration used for building Tcl. The main variations include choice of

- Building the Tcl interpreter as a static library linked into the main executable, or as a shared library that will be loaded by the executable at runtime.
- Storing the Tcl support scripts in regular files on the native file system, or within a ZIP archive that is embedded in the main executable (in the case of static linking) or shared library (in the case of shared libraries).

There are thus four main combinations possible when building Tcl with the default configuration for Tcl 9 being shared libraries with embedded ZIP archives.

From a script writer's perspective, the build configuration is more or less irrelevant but there are considerations for building single file applications for ease of deployment where build configurations matter (Section 25.3.2).

2.1.3.3. Building on Unix-like platforms

On Unix-like systems,

- Extract the downloaded tcl*VERSION*-src.tar.gz archive into a directory, say `tclsrc`.
- Run the following commands in the shell

```
mkdir tclsrc/unix/build
cd tclsrc/unix/build
../configure
make
make install
```

The above assumes you want to install Tcl in the system default location. Some commonly used options to `configure` are shown in Table 2.1.

Table 2.1. Configure options

Option	Description
`--help`	Print help text for all supported options.
`--disable-shared`	Build the statically linked version of the Tcl interpreter. A shared library is built by default.
`--disable-zipfs`	Do not embed the Tcl support library scripts into the Tcl executable. The scripts are embedded by default.
`--enable-symbols`	Build with symbols and optimizations off for debugging.
`--prefix=`*`INSTALLPATH`*	Specifies the directory where Tcl should be installed.

Next, build the Tk extension following similar steps.

- Extract tk*VERSION*-src.tar.gz into a directory, say `tksrc`, residing at the same level as the `tclsrc` directory.
- Run the commands in the shell

```
mkdir tksrc/unix/build
cd tksrc/unix/build
../configure --with-tcl=../../tclsrc/unix/build
make
make install
```

The `--with-tcl` option points to the location of the Tcl library. As with Tcl, additional options may be specified to `configure`.

2.1.3.4. Building on Windows

To build on Windows using Microsoft's compiler tool chain,

- Start the Visual Studio prompt for 32- or 64-bit builds as desired.
- Extract tcl*VERSION*-src.zip into a directory, say `tclsrc`.
- Run the commands

```
cd tclsrc\win
nmake /f makefile.vc INSTALLDIR=C:\Tcl
nmake /f makefile.vc INSTALLDIR=C:\Tcl install
```

This assumes you want Tcl installed under the `C:\Tcl` directory.

The build configuration can be controlled by passing the `OPTS` option to `nmake`. For example,

```
nmake /f makefile.vc OPTS=pdbs,noembed INSTALLDIR=C:\Tcl install
```

This enables generation of debug information and will disable embedding of the Tcl library scripts as described in Section 2.1.3.2. Similarly, adding `static` will build a static library linked into the `tclsh` shell. All available options are documented in `makefile.vc`.

To build and install Tk,

- Extract tk*VERSION*-src.zip into a directory, say `tksrc`, at the same level as `tclsrc`.
- Run the commands

```
cd tksrc\win
nmake /f makefile.vc TCLDIR=../../tclsrc INSTALLDIR=C:\Tcl
nmake /f makefile.vc TCLDIR=../../tclsrc INSTALLDIR=C:\Tcl install
```

This will build and install Tk and the GUI shell `wish`.

2.1.3.5. Building on macOS

The process of building Tcl on macOS is similar to that for Unix. Full instructions are provided in the `README` file in the `macosx` directory in the Tcl source distribution.

2.1.4. Reference documentation

The reference documentation[8] for Tcl includes includes reference pages for Tcl as well as the core packages like Tk and TDBC.

On Unix systems, the Tcl reference documentation is also available in the form of man pages accessible via the standard Unix `man` program.

On Windows, documentation location is dependent on the distribution in use. It is commonly available through an entry in the `Start` menu.

2.2. Running a Tcl program

Convention dictates we must begin our Tcl journey by greeting the world. Create a file `hello.tcl` with the following content.

```
puts "Hello World!"
```

At your shell or DOS command prompt, run this program using `tclsh` as shown below

```
C:\temp>tclsh hello.tcl
Hello World!
```

Depending on the specific distribution, your Tcl shell may have a different name such as `tclsh90`, `tclsh9.0` etc.

You have now written your first Tcl program. Feel free to add Tcl to your resumé.

2.2.1. The Tcl library and interpreter

We need to take a moment now to distinguish between Tcl, the Tcl interpreter, the Tcl library, Tcl programs or scripts, and Tcl applications.

- Tcl is the programming language. A Tcl program or script is a sequence of commands or program statements written in Tcl.
- The Tcl interpreter is a virtual machine that provides the runtime environment for running Tcl programs. As we lay out in great detail in Chapter 23, an application may contain multiple such interpreters.
- The Tcl library is the implementation of the Tcl interpreter and may be statically linked or loaded as a shared library into any application. An application makes calls into the library to create Tcl interpreters and execute Tcl programs.
- A Tcl application is a program that is written in C, or some other language, that compiles to machine code and links to the Tcl library. In some cases, the application may do very little other than provide a means to execute a Tcl script. In such cases, the Tcl script itself implements the entire functionality of the application. In other cases, the application

[8] https://www.tcl-lang.org/man/tcl/contents.htm

may natively implement much of the user visible functionality and the embedded Tcl interpreter acts as a means to allow end user scripting of the application.

If you are new to programming, no need to worry about all these terms. It is just a prelude to introducing two applications that are part of Tcl distributions — `tclsh` and `wish`.

2.2.2. The tclsh command-line shell

We have already come across the `tclsh` application briefly earlier. We now go into its functionality in some detail.

2.2.2.1. Running tclsh interactively

When run with no arguments, `tclsh` runs in *interactive mode* in the terminal or console, executing Tcl commands entered by the user and printing the result. The program terminates when the user enters the `exit` command or closes the terminal. This is commonly known as a Read-Eval-Print-Loop (REPL). A sample session is shown below.

```
C:\temp>tclsh
% puts "Hello World!"
Hello World!
% exit
C:\temp>
```

In interactive mode, `tclsh` has a few changes in behaviour, described in the following sections, vis-a-vis running a script stored in a file. These differences should be kept in mind when experimenting at the command line and then transcribing the commands to a script.

Some of these interactive features are actually implemented by a handler run when a command name is not recognized (Section 3.5.1.1).

Startup scripts: `.tclshrc`, `tclshrc.tcl`

When starting up in interactive mode, `tclsh` checks for the existence of a file `.tclshrc` (`tclshrc.tcl` on Windows) in your home directory. If found, the contents of the file are evaluated as a Tcl script before `tclsh` displays its command prompt. This can be used to load frequently used packages, define command aliases and perform other customization.

Command abbreviations

`Tclsh` will accept abbreviations for commands entered interactively as long as there is no ambiguity. For example, the Tcl command `puts` can be abbreviated as `pu` as shown here.

```
% pu "Hello!"
→ Hello!
% proc print_hello {} {pu "Hello!"}
% print_hello
→ invalid command name "pu"
% p
→ ambiguous command name "p": package pid print_hello proc puts pwd
```

Notice that abbreviations are **not** accepted if used inside a procedure (`print_hello` in our example) or if the abbreviation does not uniquely identify a command.

Execution of external programs

If the line entered by the user is not recognized as a Tcl command, Tcl will attempt to locate and run a program of that name (Section 20.1). This feature only applies to commands entered by the user, not those run as a consequence of executing the user's command and can be disabled by setting the variable `auto_noexec` to any value.

```
% uname -mr
5.15.167.4-microsoft-standard-WSL2 x86_64
% set auto_noexec ""
% uname -mr
invalid command name "uname"
```

Command history

In interactive mode, `tclsh` maintains a list of previously executed commands tagged with a *history event number*. These can be recalled at the interactive prompt using syntax similar to that used in the Unix C shell.

The `!!` form prints the previous command and executes it again.

```
% puts foo
→ foo
% !!
→ puts foo
  foo
```

The `^OLD^NEW` form replaces any occurences of `OLD` in the previous command with `NEW` and re-executes it.

```
% ^foo^bar
→ puts bar
  bar
```

The `!N` form re-executes the command tagged with the history event number `N`.

```
% history
     1  puts foo
     2  puts foo
     3  puts bar
     4  history
% !3
puts bar
bar
%
```

Also notice from the sample that you can print the list of commands executed with the `history` command. We will look at this command in more detail in Section 14.5.

Command-line editing

The `tclsh` shell does not itself include any facilities for command line editing, tab completion or cursor keys for history recall.

In a Windows environment, the DOS console already provides most of these features except tab completion. On Unix platforms, there are several alternatives that can be used:

- Use the `rlwrap` program or its equivalents[9].
- Load the `tclreadline` [10] extension from your `.tclshrc` file.
- Source the pure Tcl `tclreadline` [11] script which implements most of `tclreadline` extension's functionality.

The best option for interactive use may be to use one of the graphical shells, either the one built into `wish` or `tkcon`. In addition to line editing and tab completion the latter includes many other useful facilities.

Detecting interactive mode

Programs that needs to change their behavior depending on whether `tclsh` is running in interactive mode can check the `tcl_interactive` global variable. This is set to `1` when running in interactive mode and `0` otherwise. The program may even choose to modify its value to enable interactive behaviors (Section 20.2.1).

2.2.2.2. Running programs with tclsh

To run a Tcl application from a file, pass the file path to `tclsh` as a command-line argument (or to `wish` if it is a GUI application). The general form of `tclsh` for running scripts in a file is

```
tclsh ?-encoding ENCODING? PATH ?ARG …?
```

Here *`PATH`* is the path to the file containing the application. The `-encoding` option allows you specify the character encoding (Section 9.1) for the file. This defaults to UTF-8.

In Tcl 8, the default for the `-encoding` option is the system encoding. For portability reasons, it is best to use plain ASCII for all Tcl scripts and use the Unicode escape sequences from Table 3.1 for non-ASCII characters. Since ASCII is compatible with all encodings, this ensures compatibility irrespective of system encodings and Tcl versions.

`Tclsh` executes every command in the script and exits after the last command. There are of course facilities for terminating the script or for keeping the application running. The script may also pull in additional scripts stored in other files via the `source` (Section 3.14) command.

Additional arguments to `tclsh` are treated as program arguments to the script (Section 2.3.1).

[9] https://wiki.tcl-lang.org/page/rlwrap
[10] https://github.com/flightaware/tclreadline
[11] https://wiki.tcl-lang.org/page/Pure%2Dtcl+readline2

2.2.3. The wish graphical shell

The `wish` application is a “windowing shell”. Like `tclsh` (Section 2.2.2), it provides a wrapper for executing Tcl scripts. The difference is that `wish` is targeted towards applications with graphical user interfaces and includes the Tk extension.

Like `tclsh`, `wish` may also be named slightly differently, for example `wish90` or `wish9.0`, depending on the specific Tcl distribution.

As our book is about Tcl the language, and not GUI programming with Tk, we will only briefly describe `wish`. Our primary motivation for referencing it here is that provides an interactive environment for Tcl that has some benefits over `tclsh`.

2.2.3.1. Running wish interactively

Like `tclsh`, when invoked without any arguments, `wish` runs in interactive mode and first checks for the existence of a file `.wishrc` (`wishrc.tcl` on Windows) in your home directory. If found, its contents are evaluated as a Tcl script. The special interactive mode behaviours listed for `tclsh` also apply to `wish`.

Running `wish` on Windows

On a Windows system, starting `wish` will bring up the two windows as shown in Figure 2.1.

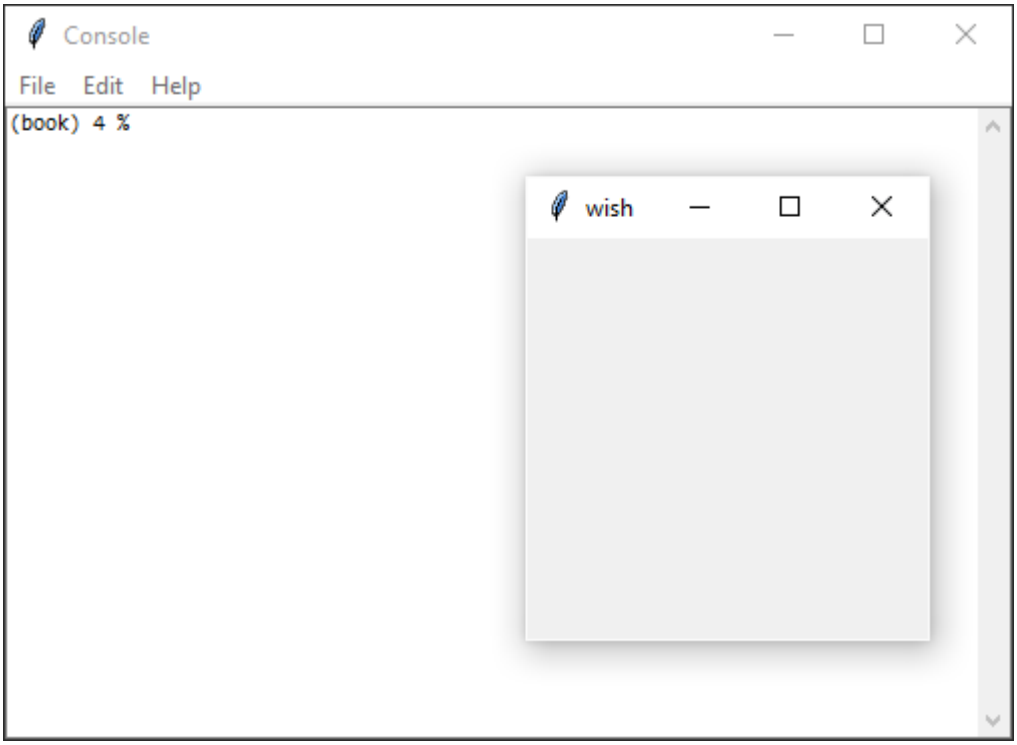

Figure 2.1. The wish windowing shell

The window titled `wish` is a toplevel window where you can add graphical elements using the Tk extension. The window titled `Console` is a Tcl command console where you can type in Tcl commands. For example, typing our usual

```
puts "Hello World!"
```

will output that line to the console window. Or typing the commands

```
ttk::label .l -text "It is easy to create interfaces in Tcl/Tk."
ttk::button .b -text Exit -command exit
grid .l .b -padx 5
```

will create a label and button arranging them as shown in Figure 2.2.

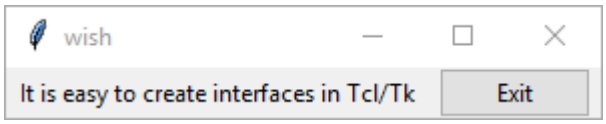

Figure 2.2. A sample Tk window

Clicking on the button will end the program. Tk makes it amazingly easy to create graphical user interfaces. Sadly, we do not have space in this book to cover it and refer you to one of the many books that do.

Running `wish` on Unix systems

Running the `wish` shell on Linux and Unix systems has different behaviour from Windows as they do not distinguish between "console" mode and "GUI" mode. Only one windows is created — the `wish` toplevel. The Tcl console will continue to be displayed in the terminal window just as for `tclsh`. You can type commands in the terminal window in the same manner as you did above to display our sample Tk window.

2.2.3.2. Running scripts with wish

Like `tclsh` (Section 2.2.2), `wish` can be passed the name of a file containing a Tcl script.

```
wish ?OPTIONS? PATH ?ARG …?
```

The contents of *`PATH`* are executed as a Tcl script with any additional arguments being passed to the script as for `tclsh`. There are additional options that may be specified but we will not cover them in this book.

There is one important difference between `tclsh` and `wish` when it comes to execution of scripts. Unlike `tclsh`, `wish` does **not** exit when the last command in the script is executed. It starts running the event loop (Chapter 19), waiting for user interaction and other events.

2.2.4. The tkcon enhanced shell

Unless you are writing graphical user interfaces, there is only one reason to use `wish` for interactive development instead of plain old `tclsh` and that is the Tk Enhanced Console `tkcon`. This is an add-on that comes as a single file `tkcon.tcl` and is included in most Tcl binary distributions.

The `tkcon` console sports a number of useful features not natively available in either `tclsh` or `wish` including enhanced command-line editing, tab completion, incremental history search, multiple interpreters, remote interpreters, hot error links and more.

You can start a `tkcon` console by passing it as the argument to `wish`.

```
wish tkcon.tcl
```

This will bring up a command-line window where you can interactively execute Tcl.

2.2.5. Exiting a Tcl application

The `exit` command terminates the Tcl application or shell where it is invoked.

```
exit ?CODE?
```

`CODE`, defaulting to `0`, is passed back to the operating system as the process exit code. We will have more to say on exit codes in Chapter 20.

2.2.6. Making Tcl scripts executable

As we have seen, any file may be executed as a Tcl script by passing it to a Tcl shell as an argument. However, it is convenient to be able to just type the script file name and have it executed. Thus we would rather execute a script by typing

```
myscript
```

as opposed to

```
tclsh myscript
```

The method for doing this differs between operating systems.

2.2.6.1. Executable scripts on Unix

On Unix, any Tcl script that is intended to be an application (as opposed to a library) should be marked as executable (via `chmod +x`) and begin with the following line or equivalent.

```
#!/usr/bin/env tclsh
```

Then assuming the `tclsh` executable lies in a directory somewhere on the path, just typing the name of the script file at the Unix shell prompt suffices to have it executed as a Tcl script.

Note that this line is treated as a comment by Tcl as it begins with a `#` character. Thus although this technique will not work on Windows, nor does it cause any harm if the same script is passed as an argument to `tclsh` on Windows.

2.2.6.2. Executable scripts on Windows

On Windows, making a script directly executable is more involved. Luckily, installers for binary distributions do these steps for you so you don't have to. If you do build and install from sources, follow the steps here.

Windows file execution from the console is based on the file extension. It is not possible to mark individual files as executable. So we need to pick a file extension to associate with Tcl. Following the Magicsplat distribution, we will associate the extension `.tclapp` with Tcl applications that can be executed directly leaving `.tcl` to be used with secondary support files and libraries.

As a second step, the extension has then to be mapped to a file type which can be any text that does not conflict with that set up by other applications. We will imaginatively call the type `TclApp`.

Because they modify the Windows registry, the commands below have to be run with elevated administrative privileges.

These two steps are accomplished with the `assoc` and `ftype` Windows commands.

```
C:\temp>assoc .tclapp=TclApp
C:\temp>ftype TclApp=C:\Tcl\bin\tclsh.exe "%1" %*
```

(Assuming that is where our Tcl is installed.)

If you type `myscript.tclapp` at the DOS command-line, Windows will invoke `tclsh` to run your script. If you want to avoid having to type the `.tclapp` extension, an additional step is needed. The `.tclapp` extension needs to be added to the list of extensions in the `PATHEXT` environment variable.

```
C:\temp>set PATHEXT=%PATHEXT%;.tclapp
```

Now just typing `myscript` is sufficient to invoke `tclsh` to execute the `myscript.tclapp` file.

You can embed Tcl scripts into Windows `.BAT` batch files using a trick[12] similar to that for Unix. However, the author prefers the method described above for a couple of reasons. First, a batch file involves execution of an intermediate Windows command shell and is therefore slower. Second, there are various scenarios where the suggested `.BAT` solutions do not work.

[12] https://wiki.tcl-lang.org/page/DOS+BAT+magic

2.3. The application runtime environment

An application's runtime environment includes

- arguments passed on the command-line
- the process environment such as working directory, environment variables etc.
- information about the Tcl interpreter itself such as version information
- platform information such as operating system, architecture and user context

2.3.1. Command-line arguments

Any additional arguments supplied on the command-line when invoking `tclsh` or `wish` are passed to the script in the global variables shown in Table 2.2.

Table 2.2. Command-line argument globals

Name	Description
`argv0`	Contains the path to the script file passed on the command line. If `tclsh` was invoked without any arguments, this will contain the name by which it was invoked (which is not necessarily `tclsh` in the presence of links etc.)
`argv`	List containing the command-line arguments.
`argc`	Count of command-line arguments.

Let us illustrate with a simple example. Create a file `reverse.tcl` with the following content which will simply reverse and print each argument.

```
# reverse.tcl
if {$argc == 0} {
    puts "Need to provide at least one argument"
    puts "Usage: [info nameofexecutable] $argv0 arg ?arg ...?"
    exit 1
}

proc print_reversed {str} {
    puts [string reverse $str]
}

foreach arg $argv {
    print_reversed $arg
}
```

This example also introduces some very basic syntax:

- Variable values are referenced by prefixing the variable name with `$`.
- Procedures are defined using `proc` and invoked like any built-in command.

The script is executed by passing it to `tclsh`. In this initial run no arguments are passed and hence `argc` is 0 resulting in the script exiting with an error message. Given arguments, the script runs to completion and exits implicitly at the end of the file.

```
C:\demo> tclsh reverse.tcl
Need to provide at least one argument
Usage: c:/tcl/866/x64/bin/tclsh.exe reverse.tcl arg ?arg ...?
C:\demo> tclsh reverse.tcl abc def
cba
fed
```

2.3.2. The working directory: pwd, cd

The `pwd` command returns the current working directory for the process.

```
% set currentDirectory [pwd]
→ C:/TEMP/book
```

The `cd` command changes the current working directory to that specified.

```
cd ?DIRNAME?
```

If the optional *`DIRNAME`* argument is not present, the command changes the working directory to the home directory of the current user.

```
% cd ❶
% pwd
→ C:/Users/apnad/Documents
% cd $currentDirectory ❷
```

❶ Change to the home directory
❷ Change back to the directory we saved in `currentDir`

The application may have multiple threads and multiple Tcl interpreters in each thread. The working directory is a process-wide setting and therefore the `cd` command affects **all** interpreters and threads in the process.

2.3.3. Environment variables: env

The environment variables for the current process are accessible through the `env` global array and can be accessed the same way as any other array variable.

```
% puts $env(PATH)
→ C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.39.3...
% array names env
→ OneDriveConsumer HOME JDK_HOME COMSPEC PROCESSOR_IDENTIFIER WindowsSdkDir Pro...
```

Arrays are fully described in Section 3.6.8.

There are however a few differences that distinguish `env` from normal Tcl arrays. First, any changes to the `env` array are automatically reflected back in the process environment. Any child processes will inherit the modified environment.

The second is that on **Windows platforms only** `env` keys are **not** case-sensitive. So

```
puts $env(hOmE) → C:\Users\apnad\Documents
puts $env(HOME) → C:\Users\apnad\Documents
```

would work as well unlike for normal arrays. Note however, array commands that accept wild card patterns are case-sensitive as illustrated by the following:

```
array names env pat* → (empty)
array names env PAT* → PATHEXT PATH
```


Because of the need to keep the `env` array synchronized with the process environment, access to its elements is orders of magnitude slower than a normal array variable. It is often beneficial to keep a "shadow" copy of the environment in a normal variable in any performance sensitive code.

2.3.4. The process identifier: pid

The `pid` command returns the process identifier, or PID, for the current process.

```
pid ?CHANNEL?
```

Without arguments, the command returns the PID of the current process.

```
pid → 19776
```

If the *`CHANNEL`* argument is specified, it must be the channel associated with a process pipeline (Section 20.2) and the command returns the list of PID's for the processes in the pipeline.

2.3.5. Executable file path: info nameofexecutable

The `info nameofexecutable` command returns the path to the executable image for the current process hosting the Tcl interpreter.

```
info nameofexecutable → c:/tcl/magic/bin/tclsh90.exe
```

Most commonly this is used to find other locations in the file system that are relative to the executable or spawn off child processes to run other Tcl applications.

2.3.6. Tcl version information: info tclversion|patchlevel

The `tcl_version` global variable contains the version of the Tcl library in use, and in effect the version of Tcl. The same information is also available with the `info tclversion` command.

```
puts $tcl_version → 9.0
info tclversion   → 9.0
```

The `tcl_patchlevel` global variable and `info patchlevel` command include the *patch level*.

```
puts $tcl_patchLevel → 9.0.1
info patchlevel      → 9.0.1
```

Version numbers in Tcl have a specific syntax and associated semantics. See Section 17.3.2.

2.3.7. Tcl configuration: tcl::pkgconfig, tcl::build-info

The `tcl::pkgconfig` command returns additional information about the Tcl configuration and build environment with some overlap with that available in the `tcl_platform` array.

The command has two subcommands. The first, `list`, returns a list of keys each of which represents a piece of configuration information. The second subcommand, `get`, is used to retrieve the value associated with a key. For example,

```
% tcl::pkgconfig list
→ debug threaded profiled 64bit optimized mem_debug compile_debug compile_stats
   ↳ libdir,runtime bindir,runtime scriptdir,runtime includedir,runtime
   ↳ docdir,runtime dllfile,runtime libdir,install bindir,install scriptdir,install
   ↳ includedir,install docdir,install
% tcl::pkgconfig get bindir,runtime
→ D:\src\tcl-installer\win\dist\tcl9\x64\bin
```

The key names printed above should be self explanatory.

The directory names returned from `tcl::pkgconfig` are not very useful. In particular, even for the keys suffixed with `runtime` as above, they reflect the values configured as runtime directories at **compile** time, and not the actual **runtime** values.

The `tcl::build-info` command returns information that is useful for developers to clone the user's Tcl application build when reproducing issues encountered in the field.

```
% tcl::build-info
→ 9.0.1+10a450bde9d304cbb6a4c2fa54ceaeea7de025dae402aff4c2884a2cce2ce595.msvc-1...
```

The returned string consists of a version number followed by an optional component prefixed with a `+` sign. This optional component consists of a sequence of tags separated by `.` characters. The first of these is the repository commit id from which Tcl was built. This is followed by a alphabetically sorted list of tags that provide additional information about how Tcl was built such as compiler version, build system, vendor patches and so on.

2.3.8. Platform information

The `tcl_platform` global array contains various bits of information about the hosting platform. The elements of this array are shown in Table 2.3.

Table 2.3. Platform information

Element	Description
`byteOrder`	`littleEndian` or `bigEndian` as per the CPU architecture.
`engine`	Identifies the interpreter implementation. This is normally `Tcl` but may hold other values if you are using other dialects such as `jim`.
`machine`	CPU architecture this executable was built for. This is not necessarily the native architecture of the system. For example, 32-bit binaries on a 64-bit Windows system will return `intel` and not `amd64`.
`os`	The operating system.
`osVersion`	The operating system version.
`pathSeparator`	The character used to separate entries in the `PATH` environment variable.
`platform`	The operating system family.
`pointerSize`	`4` for 32-bit architectures and `8` for 64-bit ones.
`threaded`	`1` if threads are enabled in Tcl and `0` otherwise. Not present in Tcl 9 where threads are always enabled.
`user`	The user account under which the process is running.
`wordSize`	The number of bytes in the C type `long` for the current architecure.

We can print the contents of the array with the `parray` (Section 3.6.8.8) command.

```
% parray tcl_platform
→ tcl_platform(byteOrder)     = littleEndian
  tcl_platform(engine)        = Tcl
  tcl_platform(machine)       = amd64
  tcl_platform(os)            = Windows NT
  tcl_platform(osVersion)     = 10.0
  tcl_platform(pathSeparator) = ;
  tcl_platform(platform)      = windows
  tcl_platform(pointerSize)   = 8
  tcl_platform(user)          = ashok
  tcl_platform(wordSize)      = 4
```


Although the `tcl_platform` array provides some information about the underlying operating system and architecture, the information is not specific enough to distinguish platforms for purposes like loading shared libraries from a common area. The `platform` package (Section 17.7) addresses this requirement.

3

Tcl Basics

This chapter lays the foundation for the rest of the book. It describes Tcl syntax, how Tcl parses and executes commands, and the use of variables and procedures.

Conceptually[1], execution of Tcl code occurs in two phases:

- The Tcl source code is parsed using some simple syntactic rules to break it up into a list of commands and their arguments.
- The commands are then invoked with the associated arguments.

The two phases may be interleaved in the sense that parsing a command may involve parsing and execution of nested commands and substitution of variables.

We will start off with the syntax of the language in the next section and then move on to describing some basic commands and program elements.

3.1. Basic syntax

The formal syntax rules for Tcl are defined in the Tcl manual[2], often called the *dodekalogue* as it is made up of 12 rules. Here we informally describe the syntax.

A Tcl *program* or *script* is a sequence of commands. The commands are separated by newline or semicolon characters that are not escaped (Section 3.2.1) or quoted (Section 3.3). In the special case of command substitution (Section 3.2.3), where nested commands are enclosed in square brackets, the trailing `]` character also terminates commands.

A command in turn is a sequence of *words*. Words are separated by space or tab characters that are not escaped or within double quotes or braces. Spaces and tabs can be included as part of a word by escaping them with a `\` or placing them within a quoted string. The line

```
puts -nonewline Hello ; puts " World!"
```

contains two commands separated by a semicolon. The first contains three words `puts`, `-nonewline` and `Hello` while the second contains two words: `puts` and a second word consisting of a space followed by `World`. The space preceding the `W` is not a word separator as it is within double quotes. If you are coming from another language, note that **simple strings like `Hello`, with no whitespace or syntactically special purpose characters like brackets or backslashes, need not be quoted**.

[1] In practice, Tcl scripts are converted to byte code form before execution
[2] https://www.tcl-lang.org/man/tcl/TclCmd/Tcl.htm

A word may also be a bracketed command whose runtime result forms the value of the word. The command below has three words, `set`, `time` and the bracketed command (Section 3.2.3) `clock seconds`.

```
set time [clock seconds]
```

A word may be spread over multiple lines when quoted with double quotes or braces, or when it comprises a bracketed command. The command below consists of **two** words, `puts` which is the command name and its argument which is quoted, with braces in this case. The quoting allows spaces and newlines to be considered part of a single word.

```
puts {
    Hello
    World
}
→
      Hello
      World
```


You can use `info complete` to check if a given string syntactically constitutes one or more complete commands.

```
info complete {foo bar "x y z}  → 0
info complete {foo bar "x y z"} → 1
```

The command does **not** check if the command names are valid, have the correct number of arguments and so on. It only checks whether the given argument can be parsed **syntactically** as a sequence of complete commands. Our first example above fails because of unmatched quotes.

The `info complete` command is mostly used with an interactive command loop as in `tclsh` to check if the user has entered a complete command.

3.2. Substitutions

Tcl performs a series of substitutions before a command is executed:

- Backslash substitutions
- Variable substitutions
- Command substitutions

These substitutions do not take place for strings enclosed in braces `{}` where different rules apply as we will describe in Section 3.3.2.

3.2.1. Backslash substitutions

The primary purpose of backslash substitution, or backslash escape sequences, is to allow arbitrary characters to be represented using a character sequence starting with a `\` character. They are primarily used for inclusion of ASCII control characters and Unicode code points (Section 4.1.1) in script text. Table 3.1 lists the format of these escape sequences.

Table 3.1. Backslash sequences

Sequence	Description
`\a`	Audible alert (ASCII 7)
`\b`	Backspace (ASCII 8)
`\f`	Form feed (ASCII 12)
`\n`	Newline / linefeed (ASCII 10)
`\r`	Carriage return (ASCII 13)
`\t`	Tab (ASCII 9)
`\v`	Vertical tab (ASCII 11)
`\`*`OOO`*	One to three octal digits specifying a Unicode code point in the range U+000000 - U+0000FF.
`\x`*`HH`*	`x` followed by one or two hexadecimal digits specifying a Unicode code point in the range U+000000 - U+0000FF.
`\u`*`HHHH`*	`u` followed by one to four hexadecimal digits specifying a Unicode code point in the range U+000000 - U+00FFFF.
`\U`*`HHHHHHHH`*	`U` followed by one to eight hexadecimal digits specifying a Unicode code point in the range U+000000 - U+10FFFF.

For example,

```
puts a\tb       → a       b
puts \351       → é
puts \xe9       → é
puts \u00e9     → é
puts \U000000e9 → é
```

Backslash sequences are also used for continuing commands across multiple lines. A `\` followed by a newline character and any amount of whitespace on the following line is replaced by a single space character.

```
% puts "abc\
                        def"
→ abc def
```

Note a single space is separating the output words above.

Finally, if the character sequence following a `\` does not fall into one of the above categories, the backslash and the following character are replaced by that character. The common use

of this is to prevent the Tcl parser from interpreting special meaning for characters such as space, `$` or the backslash itself.

```
puts a\\nb → a\nb ❶
puts \$foo → $foo ❷
puts \s    → s ❸
puts \xz   → xz ❹
```

❶ `\\` treated as a single ordinary `\`, not a `\n` sequence
❷ `$` treated as itself, not as a variable substitution
❸ No backslash sequence corresponding to `s`
❹ `\x` not followed by hexadecimal digits

3.2.2. Variable substitutions

The second type of substitution that takes place prior to the execution of a command is replacement of variable references by their values. Although variable references can take many forms, here we only illustrate the simplest one of the form `$VARNAME`.

```
set greeting "Hello World!" → Hello World! ❶
puts $greeting              → Hello World! ❷
```

❶ `set` assigns a value to a variable. The variable name is not prefixed with a `$`.
❷ Substitution of a variable name with its value requires the name to be prefixed with `$`.

Variable substitution takes place **inside** words as well unless the word is quoted with braces.

```
set greeting Hello     → Hello
set who World          → World
puts "$greeting $who!" → Hello World!
```

Tcl will raise an error if the referenced variable does not exist.

```
% puts $nosuchvar
Ø can't read "nosuchvar": no such variable
```

You can prevent Tcl from treating `$` as variable reference by prefixing it with a `\` character.

```
puts $greeting  → Hello
puts \$greeting → $greeting
```

There are two important points to note about variable substitution. The value that is substituted is **not** subject to further reparsing and substitution. Secondly, substitution of variables does not change the word boundaries in a command. In the next example, the space character in the substituted value `Hello World!` does not act as a word separator. The second command still contains only two words, the second word being the entire contents of `greeting` including the space.

```
set greeting "Hello World!" → Hello World!
puts $greeting              → Hello World!
```

The other forms of variable references (Section 3.6) are also subject to substitution as above. Note however, that a `$` character by itself, or one that is followed by a character other than an alphanumeric, underscore (`_`), left parenthesis (`(`) or left brace (`{`), is **not** a variable reference and will be treated as a literal `$` character.

```
puts $$ → $$
puts $= → $=
```

3.2.3. Command substitutions

The final form of substitution is the replacement of strings delimited by `[]` brackets with the result of evaluating them as scripts.

```
set i 0       → 0
puts [incr i] → 1 ❶
```

❶ The `incr` command increments a variable.

In the above example, the `puts` command gets a single argument — the result of `incr i`.

As for variables, command substitution takes place inside words as well.

```
puts a[incr i]b                         → a2b
puts "Incrementing $i gives [incr i]." → Incrementing 2 gives 3.
```

The string inside the `[]` pair is a Tcl script and may have multiple commands separated by semicolons or newlines. The substituted value is the return value from the last command.

```
% puts [incr i; incr i; incr i]
→ 6
% puts [
      set j 10
      incr i $j
     ]
→ 16
```

Moreover, the bracketed string is parsed independent of the outer script so it can itself contain quotes, substitutions etc. with no interference between it and the containing script. For example, the double quote following the `expr` below **starts** a quoted string within the bracketed command, it does not terminate the double quotes that follow the `puts`.

```
puts "The total is [expr "2+4"]" → The total is 6
```

As for variable substitution, command substitution does not reparse the returned value from the command string or change the word boundaries even when the substituted value contains whitespace or other special characters.

3.3. Quoting

Quoting is a means of telling the Tcl parser to treat a sequence of characters as a single word even if it contains word or command terminating characters. We have already seen one mechanism to prevent special interpretation of characters — backslash sequences. Quoting provides an alternative and more convenient means for the same. Compare the following for assigning a string containing spaces to the variable `var`:

```
% set var This\ is\ a\ single\ word
→ This is a single word
% set var "This is a single word"
→ This is a single word
```

Strings may be quoted by either

- enclosing the string in double quotes, or
- enclosing the string in braces.

The two differ in how substitutions are handled.

3.3.1. Quoting using double quotes

When a string is enclosed in double quotes, word and command separators like spaces, tabs, newlines and semicolons are treated as ordinary characters. Spaces within the quoted string are ignored as word separators and the semicolon and newline do not terminate the command.

```
% puts "This is line one;
This is line two"
→ This is line one;
  This is line two
```

The double quote character only has effect at the beginning of a word. It is otherwise treated as an ordinary character. The closing double quote must be followed by a word separator or command terminator. Thus the following result in errors.

```
% set var foo"b ar" ❶
Ø wrong # args: should be "set varName ?newValue?"
% set var "foo"bar  ❷
Ø extra characters after close-quote
```

❶ `foo"b` and `ar"` are separate words since quotes within a word carry no special meaning.
❷ Closing quote is not followed by a word separator.

A double-quoted string is subject to *string interpolation* wherein backslash, variable and command substitutions take place. To prevent special treatment of characters including a literal double quote, escape it with a `\`.

```
% puts "$i\n[incr i]"
→ 16
  17
% puts "\$i\n\[incr i]" ❶
→ $i
  [incr i]
% set var "foo\" bar" ❷
→ foo" bar
```

❶ Does not substitute `$` or `[incr i]`.
❷ Literal double quote in string.

> **!** Literal braces must also be escaped inside double-quoted strings. This is being specifically called out because you may find escaping unnecessary when experimenting on an interactive command line. However, this will fail when the double-quoted string is nested in a brace-quoted blocks.

3.3.2. Quoting using braces

In the second form of quoting, the quoted string is enclosed in a pair of braces `{}`. With the exception noted below, all special treatment for characters and substitutions are disabled within the enclosed string. Here is an example contrasting the two forms.

```
% puts "$i\n[incr i]" ❶
→ 17
  18
% puts {$i\n[incr i]} ❷
→ $i\n[incr i]
```

❶ Substitutions enabled inside double quotes
❷ Substitutions disabled inside braces

The sole exception to the above is when the backslash is followed by a newline.

```
% puts {abc\
              def}
→ abc def
```

As always, the `\`, newline and any immediate whitespace is replaced by a single space.

As for quoting using double quotes, quoting using braces must follow certain rules.

- The `{` must be the first character of a word.
- The `}` must be followed by a word separator or command terminator.

There is however an additional feature (or complication) with braces in that braced strings can nest so that the quoted string is terminated only when the number of closing braces matches the number of opening braces.

```
set nested {Outer {Inner Words} Words} → Outer {Inner Words} Words
```

As we shall see this nesting property is useful for defining structures like lists or dictionaries and for creating "code blocks" for conditional or iterative commands like `if` or `while`.

There is one rarely occurring idiosyncracy with regard to nesting braces and that has to with inclusion of a literal brace character within the braced string. When a brace is preceded by a `\` it does *not* count towards the nesting depth. However, because backslash substitution rules are not in effect, the `\` character is also included in the quoted string.

```
puts {abc \}} → abc \}
```

So getting a literal brace character without a preceding `\` character in a braced string is a little tricky. One alternative is to switch to double quotes instead (taking care to properly escape unwanted substitutions).

```
puts "abc \}" → abc }
```

The other option is to use an explicit string construction command such as `string cat` (Section 4.12).

3.3.3. Choosing the quoting mechanism

When interpolating strings with values in variable or computation results, use double quotes as braces will not give the desired result.

```
% puts "The current time is [clock format [clock seconds] -format %H:%M]"
→ The current time is 21:05
% puts {The current time is [clock format [clock seconds] -format %H:%M]}
→ The current time is [clock format [clock seconds] -format %H:%M]
```

Conversely, if interpolation or substitution is not desired, use braces. The most common cases are script blocks passed to Tcl commands such as `while`, `proc` etc. Other circumstances where braces are more readable include representation of nested data structures and special situations like file paths in Windows systems where `\` is also a path separator.

```
set path "C:\\Windows\\System32\\cmd.exe"
set path {C:\Windows\System32\cmd.exe}
```

Double quotes are not special inside braces and treated as literal characters.

```
set var VALUE → VALUE
puts {"$var"} → "$var"
```

3.4. Argument expansion

The final action taken before a command is executed is argument expansion. Normally every word that is parsed is passed to the command as a single argument. However, when a word is prefixed with the character sequence `{*}`, the associated word is treated as a list of words and every element of the list is passed to the command as a separate argument.

To illustrate, let us first define a command that will print the number of arguments passed to it. Do not worry about the implementation.

```
proc nargs {args} {puts [llength $args]}
```

Let us then define a list containing three elements. Lists are covered in Chapter 5. Here it suffices to know that the `list` command creates a list containing the arguments passed to it.

```
set lst [list first second third] → first second third
```

Now compare the following statements:

```
nargs $lst    → 1
nargs {*}$lst → 3
```

In the first case, the procedure is passed a single argument - `first second third`. In the second case, application of the `{*}` operator causes the list to be expanded into its constituent elements which are then passed as to the `nargs` procedure as separate arguments `first`, `second` and `third`.

Argument expansion applies no matter in what form the following word is supplied. It could be a variable as in our example, or a quoted string:

```
nargs {first second}    → 1 ❶
nargs {*}{first second} → 2
```

❶ Literal string interpreted as a list

Or a bracketed command:

```
proc cmd_returning_a_list {} { return {9 10 11} } → (empty)
nargs [cmd_returning_a_list]                       → 1
nargs {*}[cmd_returning_a_list]                    → 3
```

In all cases, the value that would be substituted is treated as a list and its elements are passed as separate arguments to the command.

You will see examples involving the expansion operator throughout this book.

3.5. Commands

> He who wishes to be obeyed must know how to command.
>
> — Machiavelli

So now that we know how the Tcl parser breaks up a script into commands that are executed in sequence, we can move on to some basics of command execution.

The term *command* is used here and in the Tcl documentation in two distinct, though related, ways. In the code fragment

```
puts "Hello World!"
```

the term command can refer to just `puts` or to `puts` together with its argument(s). In most cases, this distinction is immaterial or clear from the context. Where it matters, we will use the terms *command statement* for the latter.

3.5.1. Command invocation

After a command statement is parsed into its final form and all substitutions and argument expansions have taken place, the first word of the statement is checked for the name of a valid command. If so found, it is executed with the remaining words passed as arguments.

Here is a crucial point to note: **the interpretation of arguments, including their type and semantics, is completely up to the command**. Tcl itself does not know or care. For example, compare the following

```
puts "2 + 2"                → 2 + 2
expr "2 + 2"                → 4
regexp "2 + 2" "2 + 2 + 3" → 0
```

The `puts` (Section 13.6) command will treat its argument `2 + 2` as a string. The `expr` (Section 7.2.2) command on the other hand will treat it as an arithmetic expression. The `regexp` (Section 10.1) command will treat the first argument as a regular expression and the second as a string.

In other words, commands are completely free to treat their arguments in any manner they see fit - as strings, numerics, program code, or anything else.

Indirect invocation of commands

Recall from our earlier discussion that substitutions are also applied to the first word before it is looked up as a command name. In particular, it is possible to invoke a command indirectly through a variable. This is commonly used in callbacks and such where a command name is passed as an argument and invoked as a callback.

```
set cmd puts        → puts
$cmd "Hello world!" → Hello world!
```

3.5.1.1. Unknown command handlers

If a command is not known to Tcl, Tcl invokes a procedure called `unknown` passing it the name of the command and associated arguments. The result of this procedure is returned as the result of the original command.

The handling of unknown commands is a little different in the presence of namespaces but since we have not discussed those yet, we will defer a full discussion to Section 16.5.3.4.

Tcl provides a default implementation of `unknown` which takes the following steps:

- It searches Tcl's library paths via the `auto_load` command (Section 17.2). This step is skipped if the `auto_noload` global variable is defined.
- If not found in the above search and not running in interactive mode, an invalid command error exception is raised. In interactive mode, the following additional steps are taken.
- It calls the `auto_execok` command (Section 20.1.2.1) to try and locate an external program of that name. If found, it will run it with the `exec` command (Section 20.1) returning the output of the program as the result of the original command.
- If the above steps fail, the command is matched against the patterns for retrieval from the command history (Section 2.2.2.1). On a match the corresponding entry from the history is executed again.
- As a last resort, if the name is an abbreviation of exactly one known command (so there is no ambiguity) that known command is used in its place.
- If all the above fail, Tcl raises an exception (Chapter 15).

```
% put "Hello World!" ❶
→ Hello World!
% echo "Hello Universe!" ❷
```

❶ `put` is an unambiguous prefix of `puts`
❷ `echo` is an external command

This default implementation can be overridden by redefining (Section 3.5.5) the procedure.

```
rename unknown _old_unknown
proc unknown {args} {
    if {$::tcl_interactive && [info level] == 1} {
        if {![catch {expr $args} result]} {
            return $result
        }
    }
    error "Unknown command [lindex $args 0]"
}
```

The new definition treats any unknown command as an arithmetic expression but only in interactive mode. Don't worry about the details of its working as we have not introduced some of the features in use.

With this new definition, we can use the Tcl shell as an interactive calculator.

```
% 2 + 4*10
→ 42
```

Before we go on, let us restore the default implementation of `unknown` as we will need it later.

```
% rename unknown ""
% rename _old_unknown unknown
```

3.5.2. Comments

If the Tcl parser encounters a `#` character where it is expecting the first word of a command, the `#` and all characters till the end of that logical line are ignored.

```
# puts "This line will not print as it's commented out"
# This is a comment\
    across two lines
```

The `#` is not treated as a comment if it appears anywhere other than where the first (non-whitespace) character of a command is expected. So for example, we can print it, name a variable, or whatever.

```
puts #Hi             → #Hi
set # "Hello world!" → Hello world!
puts ${#}            → Hello world! ❶
```

❶ This uses the variable reference syntax described in Section 3.6.1

We can even define a command implemented as a procedure named `#`.

```
proc # {s} { puts $s }
```

However, the following invocation will not work because it will be treated as a comment.

```
# "Hello world!" → (empty)
```

Instead we have to call it using one of the following syntaxes.

```
\# "Hello world!"   → Hello world!
{#} "Hello world!"  → Hello world!
set name #          → #
$name "Hello world" → Hello world
```

The above examples just illustrate the point that the check for the `#` character happens before any substitutions. They are not something you will run into in real-world Tcl code.

Tcl checks for comments *after* parsing the command into words but *before* substitutions. This leads to an idiosyncracy wherein braces must be matched even within comments. The following snippet has an unmatched brace within the comment.

```
proc demo {n} {
    # This is a comment with an unmatched { character
    return $n
}
demo
```

If you place this code in a file and try to run it with the `source` (Section 3.14) command, you will get the error

```
unmatched open brace in list
```

When parsing the above script into words, Tcl encounters the left brace character. At that point, it will look for the matching closing brace, even across line boundaries. Not finding one will lead to the error.

The lesson in this? **Match your braces even within comments!** This is admittedly quirky coming from other languages but a small price to pay for Tcl's syntactic uniformity which is the root cause of this behaviour.

3.5.3. Renaming a command

```
rename OLDNAME NEWNAME
```

Tcl is a completely dynamic language where any programming construct can be added, removed or replaced at will. This applies to commands as well, including those built into Tcl. We can thus change the name of any command with the `rename` command. A common use for `rename` is to "wrap" a command to add some functionality or to modify its behaviour in some way. We saw this in Section 3.5.1.1.

As another example, if we wanted all output to be in upper case without having to modify the application itself, we could wrap the `puts` command as follows

```
% rename puts builtin_puts     ❶
% puts "Hello world!"          ❷
Ø invalid command name "puts"
% builtin_puts "Hello world!" ❸
→ Hello world!
```

❶ Save the Tcl's puts under another name
❷ Fails because there is no longer a command called `puts`...
❸ ...but there is one called builtin_puts

Then we define a new `puts` command which makes use of the original command.

```
proc puts args {
    set str [string toupper [lindex $args end]]
    builtin_puts {*}[lreplace $args end end $str] ❶
}
puts "Hello world!"
→ HELLO WORLD!
```

❶ See Section 3.4 for explanation of the `{*}` sequence

We now get all output in upper case. The above code uses commands we have not gotten to as yet but the main idea behind wrapping commands in this fashion should be clear. We transformed the data before passing it on to the original command.

Although the above method of "wrapping" commands works with `puts`, it is incomplete for general use and will not work correctly with commands whose behaviour is dependent on the Tcl call stack. We will revisit this later in Section 14.1.7.

You will find the wrapping of commands commonly used in the Tk graphical toolkit where each GUI widget instance name is also a command. One technique for extending a widget is to replace its “owning” command with one that calls the original while adding additional behaviours.

3.5.4. Deleting a command

Passing an empty string as the second argument to the `rename` command will delete the command named by the first argument. We can use this to put things back the way they were.

```
% rename puts "" ❶
% puts "Hello world!" ❷
Ø invalid command name "puts"
% rename builtin_puts puts
% puts "Hello world!"
→ Hello world!
```

❶ Get rid of our version of `puts`
❷ Fails because the command has been deleted

3.5.5. Redefining commands

In our prior example, we renamed the command before creating our own version of it because we wanted to preserve the functionality of the original. If that is not required, we can just overwrite a command implementation simply by defining a command of the same name.

Suppose you had to write a procedure that needed some expensive one time initialization. You might implement it by explicitly maintaining a flag variable that indicates whether this is the first call to the command and then checking its existence on every call.

We could write it as shown below instead, redefining the procedure **within itself** and calling it. The `tailcall` command is explained in Section 14.1.7 but not important here.

```
proc my_proc {} {
    puts "Pretend this is some expensive initialization"
    proc my_proc {} {
        puts "Now doing the real work"
    }
    tailcall my_proc
}
```

Now let us call it twice and see it in action.

```
% my_proc
→ Pretend this is some expensive initialization
  Now doing the real work
% my_proc
→ Now doing the real work
```

Our procedure has eliminated any need for an initialization check on every call. A generalized form of self initialization is given in Section 14.4.1.

Although Tcl allows redefinition of even the core commands like `set`, `proc` etc. you are strongly advised against doing so unless you **really** know what you are doing and can duplicate their **exact** behaviour.

3.5.6. Enumerating commands: info commands

```
info commands ?PATTERN?
```

The primary command used for introspection in Tcl is `info`. When passed `commands` as its first argument, it returns a list of names of commands visible in the current namespace context. Command visibility and namespace contexts are discussed in Chapter 16.

If *`PATTERN`* is not specified, the command returns the names of all visible commands. Otherwise, only those names matching *`PATTERN`* using the rules of `string match` (Section 4.24) are returned.

```
% info commands                          ❶
→ print_args tell socket subst open eof lremove pwd _SetupCawtPkgs glob list pi...
% info commands co*                      ❷
→ coroinject coroutine concat const continue coroprobe
% info commands ::tcl::mathfunc::* ❸
→ ::tcl::mathfunc::round ::tcl::mathfunc::wide ::tcl::mathfunc::isinf ::tcl::ma...
% info commands ::tcl::*::*              ❹
```

❶ All commands visible in the current namespace.
❷ All commands visible in the current namespace that start with `co`.
❸ Commands in the namespace ::tcl::mathfunc.
❹ Namespace components in the pattern are *not* treated as wildcards so this returns an empty list.

You can also use `info commands` to check for the existence of a command. For example, older versions of Tcl did not have an `lmap` command so you might see code of the form

```
if {[llength [info commands lmap]] == 0} {
    proc lmap {args} {
        # A fallback implementation of lmap
    }
}
```

If the `lmap` command existed, `info commands` would return a list containing `lmap`. If it returns a list of zero length instead, the command does not exist and the code defines a `lmap` command implemented in script.

3.5.7. Command implementation types: info cmdtype

```
info cmdtype COMMAND
```

Commands may be implemented in Tcl by several means. The `info cmdtype` command can be used to discover how a particular command is implemented.

```
info cmdtype set     → native
info cmdtype info    → ensemble
info cmdtype unknown → proc
```

The `info cmdtype` command is not present in Tcl 8.6 and earlier.

The result of the command is one of the values shown in Table 3.2.

Table 3.2. Command types

Sequence	Description
`alias`	Command alias (Section 23.6).
`coroutine`	Coroutine (Chapter 24).
`ensemble`	Namespace ensemble command (Chapter 16).
`import`	Command imported from a namespace (Section 16.5.3.1).
`native`	Native commands written in C.
`object`	A TclOO object (Chapter 18).
`private`	The private command for a TclOO object.
`proc`	Named (Section 3.5.9.1) or anonymous (Section 3.5.9.4) procedure.
`interp`	A Tcl interpreter (Chapter 23)
`zlibStream`	A zlib stream (Section 8.5)

3.5.8. Command ensembles

In Tcl, commands need not be a single word. In many cases, commands are comprised of two or more words where the first word is the primary command used to group together commands with related functionality. These grouped commands are termed *ensemble commands*. An example is the `string` ensemble which implements various operations on strings. For example,

```
string length abcd     → 4
string index abcd end  → d
string range abcd 1 2  → bc
```

Defining ensemble commands using namespaces is covered in detail in Section 16.6.

3.5.9. Procedures

Procedures allow you to define new Tcl commands at the script level. We have already seen simple examples of procedures and we now delve into them in more detail.

Procedures may be *named* or *anonymous*. We will describe the former first.

3.5.9.1. Defining procedures: proc

```
proc NAME PARAMS BODY
```

Named procedures are defined with the `proc` command. This creates a new command called `NAME` replacing any existing command of that name. `NAME` may include *namespace qualifiers* in which case the procedure is defined within the corresponding namespace context (Chapter 16). The *BODY* argument is the Tcl script that implements the command defined by the procedure. The result returned when the new command is invoked is that of the last statement executed in `BODY`. This is not necessarily the last physical statement in `BODY` but may be a `return` or other control command.

Be aware that like all built-in Tcl commands, `proc` is also just a command like any other, not a keyword that is afforded special treatment by the Tcl parser. Although by convention the parameter definitions and the body arguments are braced, there is no such requirement imposed by Tcl. The arguments to `proc` are evaluated with the same quoting and substitution rules as for any other command.

Below is an example where you want Tcl's quoting and substitution rules to come into play. The `make_adder` procedure creates *another* procedure that adds some fixed amount to the passed number.

```
proc make_adder {increment} { proc add$increment n "expr \$n + $increment" }
make_adder 10
add10 20
→ 30
```

Pay attention to the quoting and substitutions in the above and make sure you understand how it works. We will have much more to say about this type of code construction in Section 14.4 and Section 23.12.2.

3.5.9.2. Procedure parameters

The *PARAMS* argument to the `proc` command is a list of parameter definitions, each element in the list corresponding to an argument that must be passed to the command when it is called, either explicitly or via defaults in the definition. When the command is invoked, each argument is assigned to a variable with the same name as the corresponding parameter in the procedure definition.

A procedure definition may have any number of parameters, including zero.

```
proc likes {paramA paramB} {
    puts "I like $paramA and $paramB."
}
likes ham cheese
→ I like ham and cheese.
```

Passing a different number of arguments than the number of parameters in a procedure definition results in an error.

```
% likes ham
Ø wrong # args: should be "likes paramA paramB"
% likes ham cheese eggs
Ø wrong # args: should be "likes paramA paramB"
```

Note the distinction we make between *parameters* and *arguments*. The former goes with the procedure definition. The latter refers to the values passed when the procedure is called. Each argument value is assigned to the corresponding parameter at the time of procedure invocation. Parameters are also referred to as *formal arguments*.

3.5.9.2.1. Default argument values

Each parameter definition in a parameter list is a list of one or two elements. The first element is the name of the parameter. The second element, if present, is the default argument value to assign to the parameter if the caller does not supply one. Thus we can rewrite `likes` as

```
proc likes {paramA {paramB jelly}} {
    puts "I like $paramA and $paramB."
}
```

If the command invocation does not supply a second argument, the default value is passed.

```
likes "peanut butter" → I like peanut butter and jelly.
likes ham cheese      → I like ham and cheese.
```

If a parameter has a default, all succeeding parameters must also have a default specified; otherwise an error exception will be raised when the procedure is called.

3.5.9.2.2. Variable number of arguments

Some commands support an arbitrary number of operands, for example the `list` (Section 5.1) command which constructs a list containing the arguments. In other cases, commands support various options that modify the behaviour of the command. In both cases, the command implementation has to be able to deal with an arbitrary number of arguments passed to it.

If the **last** parameter in a procedure definition is named `args`, all arguments in a call to the procedure starting at that parameter position are collected into a list and passed to the procedure as the value of the `args` parameter. The list may be empty if there are no arguments starting at that position.

```
proc likes {paramA args} {
    puts "I like $paramA."
    if {[llength $args] != 0} {
        puts "I also like [join $args {, }]"
    }
}
```

In the first call below, `args` is empty. In the second, it contains `cheese`, `apples` and `bananas`.

```
% likes broccoli
→ I like broccoli.
% likes ham cheese apples bananas
→ I like ham.
  I also like cheese, apples, bananas
```

3.5.9.2.3. Named parameters and options

When a procedure has a large number of parameters, some possibly being optional, calling it can be awkward and error-prone having to remember the position of the parameters and supplying values even in the presence of defaults. For example, consider the following procedure definition:

```
proc fontify {text {family Arial} {style normal} {weight medium} {size 10}} {
    return "<span font-family='$family' font-style='$style' \
                font-weight='$weight' font-size=$size>$text</span>"
}
```

Calling this procedure for a non-default style and font size requires all arguments to be specified even though most use the default.

```
% fontify "Some text" Arial italic medium 12
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size...
```

Some languages support *named parameters* that deal with this problem by allowing a subset of the parameters to be passed by specifying them by name. Although Tcl does not have built-in named parameters, we can achieve similar results through the `args` facility as below.

```
proc fontify {text args} {
    lassign {Arial normal medium 10} family style weight size
    if {[llength $args] & 1} {
        error "No value specified for parameter [lindex $args end]"
    }
    foreach {param val} $args {
        set $param $val
    }
    return "<span font-family='$family' font-style='$style' \
                font-weight='$weight' font-size=$size>$text</span>"
}
fontify "Some text" size 12 style italic
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size...
```

Note that the order of optional arguments is immaterial as they are identified by parameter name. We initialize the local variables to defaults using `lassign` (Section 5.7) and then loop through the variable arguments overwriting the defaults. Other means of accomplishing the above include using arrays (Section 3.6.8) and dictionaries (Chapter 6). For example, using dictionaries,

```
proc fontify {text args} {
    set opts [dict merge {
        family Arial style normal weight medium size 10
    } $args] ❶
    dict with opts {} ❷
    return "<span font-family='$family' font-style='$style' \
                font-weight='$weight' font-size=$size>$text</span>"
}
fontify "Some text" size 12 style italic
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size...
```

❶ Merge default option value dictionary with provided arguments
❷ Copy dictionary elements into local variables

A variant on named parameters are *options* or command switches where the named arguments are passed prefixed with `-`, `--`, `/` etc. depending on platform and convention.

The above samples have a couple of drawbacks. Errors like misspelt parameter names are not detected and parameter introspection (Section 3.5.9.5) provides limited information. The above also does not handle boolean switches — options that do not take a value. While the examples can be adapted for this, you are best off using one of several third-party packages[3] rather than reinventing the wheel.

[3] https://wiki.tcl-lang.org/page/command+options

3.5.9.3. Returning from a procedure: return

```
return ?VALUE?
```

The syntax shown above is the simplest form of the `return` command. In its general form (Section 15.3), it is more flexible and powerful than illustrated here.

The command stops execution of a procedure, returning *`VALUE`*, which defaults to an empty string, as the procedure result.

```
proc signum {n} {
    if {$n < 0} {
        return -1
    } elseif {$n == 0} {
        return 0
    } else {
        return 1
    }
}
signum -5
→ -1
```

If a procedure falls through to the end of its body without encountering a `return`, the result of the last command executed in the body is the result of the procedure call. The `double` procedure below returns the result of `expr`.

```
proc double {n} { expr {2*$n} }
double 4
→ 8
```

3.5.9.4. Anonymous procedures: apply

```
apply ANONPROC ?ARG …?
```

There are some common idioms in programming where code is executed via a callback mechanism. Examples include callbacks used for comparing elements when sorting lists, callbacks registered as handlers for timers or I/O events etc. In these cases where a "one-off" procedure is needed that is invoked from only one call site, it is inconvenient to have to define a separate named procedure for each such use. Anonymous procedures are a more convenient alternative.

Since anonymous procedures have no name, there has to be some facility to actually invoke them. This facility is the `apply` command.

The `apply` command takes a mandatory argument — the anonymous procedure — and invokes it passing it any additional arguments that are supplied. The anonymous procedure *`ANONPROC`* itself is a list containing two or three arguments:

- the parameter definitions in the same form as for a named procedure
- the body of the procedure, again as for a named procedure
- optionally, the namespace (Chapter 16) in which the procedure is to reside.

We will illustrate with a simple example using the `lsort` command. This command is fully detailed in Section 5.21 but here it suffices to know it has an option, `-command`, whose associated value should be a callback command for comparing elements. When this callback is invoked, it is passed the elements to be compared and should return `-1`, `0` or `1` depending on whether the first element is less than, equal or greater than the second.

Let us assume we want to sort a list of integers based on their **absolute** values. We pass `lsort` the `apply` command with an anonymous procedure that does the comparisons. Note the similarity of the anonymous procedure to the structure of a named procedure definition: the parameter definitions followed by the procedure body.

```
namespace import tcl::mathfunc::abs ❶
set list_of_ints {-1 5 -5 10 -5 -1000 100}
lsort -command {apply {
    {a b}
    {
        if {[abs $a] < [abs $b]} { return -1 }
        if {[abs $a] > [abs $b]} { return 1 }
        if {$a < $b} { return -1 }
        if {$a > $b} { return 1 }
        return 0
    }
}} $list_of_ints
→ -1 -5 -5 5 10 100 -1000
```

❶ The `abs` function returns the absolute value of a number.

The above looks slightly clumsy so applications often define a helper procedure, generally called `lambda`, for constructing anonymous procedures.

```
proc lambda {params body args} {
    return [list ::apply [list $params $body] {*}$args]
}
```

Then the above sort can be written as

```
lsort -command [lambda {a b} {
    if {[abs $a] < [abs $b]} { return -1 }
    if {[abs $a] > [abs $b]} { return 1 }
    if {$a < $b} { return -1 }
    if {$a > $b} { return 1 }
    return 0
}] $list_of_ints
→ -1 -5 -5 5 10 100 -1000
```

which looks cleaner. The `lambda` looks almost exactly like a normal `proc` definition except for the missing procedure name. This construct is used so often that Tcllib[4] contains a package `lambda` that defines this `lambda` command for you[5].

Could we have written the above as a named procedure? Of course. The choice is a personal preference. Defining a named proc means one more suitable name to think of[6], potentially less clarity compared with inline code and so on.

Why procedures are preferable to scripts as callbacks

There are some advantages to passing a procedure as a callback as opposed to a script.

- The first is performance since procedures are compiled into a byte code form that is faster than that for scripts.
- The second is that any non-trivial script will use variables whose names will "pollute" the callback context. This problem does not arise with procedures where any variables will be local to the procedure.

3.5.9.5. Introspecting procedures: info procs|args|default|body

```
info procs ?PATTERN?
info args PROCNAME
info default PROCNAME PARAMNAME VARNAME
info body PROCNAME
```

While the `info commands` (Section 3.5.6) command we saw earlier returns the list of commands visible in the current context, the `info procs` command only returns names of procedures. If *`PATTERN`* is specified, it is matched using the rules of `string match` (Section 4.24).

```
info procs      → print_args _SetupCawtPkgs auto_load_index likes unknown my_proc
 nargs _InitCa...
info procs tcl* → tclPkgUnknown tclPkgSetup tclLog
```

The `info args` command returns a list containing the names of the procedure's parameters.

```
proc likes {paramA {paramB jelly}} {puts "I like $paramA and $paramB."}
info args likes
→ paramA paramB
```

The above only returned the *name* of each parameter, not the entire parameter definition. To get the defaults associated with a parameter, we have to use the `info default` command. The command returns `1`, storing the default in a specified variable, if the parameter has a default and returns `0` otherwise.

[4] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md
[5] The name `lambda` comes from Lambda Calculus and generically refers to anonymous functions.
[6] This is not a trivial problem when there are a lot of callbacks, often with similar functionality, in an application!

```
info default likes paramB paramB_default → 1
puts $paramB_default                     → jelly
```

Lastly, the `info body` command returns the entire body of the procedure.

```
% info body likes
→ puts "I like $paramA and $paramB."
```

We can use these commands to entirely reconstruct a procedure definition at run time without access to source.

```
proc reconstruct {proc_name} {
    set proc_name [uplevel 1 [list namespace which -command $proc_name]]
    set params [lmap param_name [info args $proc_name] {
        if {[info default $proc_name $param_name defval]} {
            list $param_name $defval
        } else {
            list $param_name
        }
    }]
    return [list proc $proc_name $params [info body $proc_name]]
}
```

We can then call it to reconstruct any procedure from the runtime environment.

```
% reconstruct likes
→ proc ::likes {paramA {paramB jelly}} {puts "I like $paramA and $paramB."}
```

A short explanation is in order for the commands used above that we have not covered yet. The `namespace which` command converts the supplied procedure name to a fully qualified name in case the procedure is defined in a namespace. The `lmap` (Section 5.19) command, loops through a list and constructs a new list containing the result from each iteration.

This kind of reconstruction is useful in Tcl tools like profilers and debuggers as well as in metaprogramming constructs like macros. Some Tcl packages, like `pipethread`, even use similar methods to propagate code defined in one interpreter to other remote interpreters as part of a parallel computation framework.

3.6. Variables

We have already seen basic usage of variables. It is time to go into more details including name syntax, scopes, visibility and commands related to variable management.

3.6.1. Variable name syntax

Unlike most other languages, there are practically no restrictions on the characters that can be used in a variable name. It is convenient for many reasons to restrict names to the “standard” alphanumeric plus underscore (`_`) convention but this is not mandated. You can

include practically any character you wish in a variable name as long as you take care to appropriately quote or escape it as per the Tcl parser rules.

```
set a_traditional_variable "A variable name"
set {Funky + Var # Name} "can be anything"
set "" "you like." ❶
→ you like.
```

❶ Even an empty string can be a variable name!

There are two special cases of variable names that we will detail in later sections. The first is with regard to use of parenthesis in array references (Section 3.6.8). The other is that although a colon (:) character can be used in a variable name, two or more *consecutive* colons in the name signify a variable in a namespace (Chapter 16).

3.6.2. Variable assignment: set

```
set VARNAME ?VALUE?
```

The basic command for assigning a value to a variable is `set`. If the *`VALUE`* argument is not supplied, the command returns the value of the variable named *`VARNAME`* if it exists and raises an error if it does not. If *`VALUE`* is specified, it is assigned to the variable. In all cases, the command returns the new value of the variable.

```
set avar "Some value" → Some value
set avar              → Some value
```

Strange though it may seem, there is no guarantee that the return value of the command (which is also the new value of the variable) is the specified *`VALUE`*! This is true not only for `set` but other commands that modify variables as well. This can happen when there are *traces* on the variable (Section 14.2.1).

3.6.3. Getting a variable's value

We have already seen how the value stored in a variable is retrieved either by prefixing its name with the `$` character or using the single argument form of the `set` command.

```
puts $avar      → Some value
puts [set avar] → Some value
```

If a variable name contains whitespace or other special characters, dereferencing the variable requires the name to be delimited with a pair of braces.

```
% puts "$a_traditional_variable ${Funky + Var # Name} ${}"
→ A variable name can be anything you like.
```

In some cases, braces are required to delimit variable names even for "normal" variable names. Suppose we wanted to write a procedure to pluralize a word by very simplistically appending an `s`. The following does not work because the Tcl parser will treat `$nouns` as a reference to the variable `nouns` as opposed to a reference to `noun` followed by a literal `s`.

```
proc pluralize {noun} {return $nouns} → (empty)
pluralize car                         Ø can't read "nouns": no such variable
```

To fix this, we need to delimit the variable name with a pair of braces.

```
proc pluralize {noun} {return ${noun}s} → (empty)
pluralize car                             → cars
```

Even this braced form of variable reference will not work when the name itself contains braces. In such cases, you have to resort to the `set` command to retrieve the value.

```
set "{namewithbraces}" avalue → avalue ❶
puts ${namewithbraces}        Ø can't read "namewithbraces": no such variable ❷
puts ${{namewithbraces}}      → avalue ❸
puts [set "{namewithbraces}"] → avalue ❹
```

❶ The variable name is `{namewithbraces}`, i.e. the name itself contains braces
❷ Fails because dereferences `namewithbraces`, not `{namewithbraces}`
❸ Works in Tcl 9 but not in Tcl 8 where name references are terminated at the first `}` without counting nested braces.
❹ Works in both Tcl 8 and Tcl 9

A similar situation arises when accessing the value of a variable *indirectly* through another variable that holds the name of the first. Consider the following

```
set avar "Some value" → Some value
set bvar "avar"       → avar
```

The following attempts fail to retrieve the value whose **name** is stored in `bvar`.

```
puts $$bvar   → $avar
puts ${$bvar} Ø can't read "$bvar": no such variable
```

The first does not raise an error but does not return the desired value. It illustrates an important characteristic of the Tcl parser. When a substitution is made, it will never go back and reparse the string. Thus the parser never sees the substituted string `$avar`. The second attempt fails because variables are not substituted inside the brace-quoted strings.

The single argument form of the `set` command comes into use here.

```
puts [set $bvar] → Some value
```

Tcl substitutes `$bvar` with its value `avar` which is passed to the `set` command which then returns the value of the `avar` variable.

An alternative is to use `upvar` (Section 14.1.4) to create an alias. This is more convenient if the variable is referenced multiple times.

```
upvar 0 $bvar ref → (empty)
puts $ref         → Some value
```

Needless to say, there is no reason to use weird variable names in your programs. However, the ability to do so is sometimes useful in languages like Tcl where the variables may be dynamically generated at runtime.

3.6.4. Unsetting variables: unset

```
unset ?-nocomplain? ?--? ?VARNAME …?
```

The `unset` command deletes one or more variables. The `-nocomplain` option suppresses the error that would otherwise be generated if a variable does not exist.

```
set avar "Some value"  → Some value
unset avar             → (empty)
unset avar             Ø can't unset "avar": no such variable
unset -nocomplain avar → (empty)
```

3.6.5. Variable scopes, lifetimes and visibility

A variable's *scope* is the region within which a variable is defined and can be referenced without special qualification.

Tcl defines three scopes:

- *Local* scope where a variable is defined within a procedure.
- *Namespace* scope where a variable is defined within a namespace. This is described in Chapter 16 and we will not say more about it here.
- *Global* scope where the variable is defined outside any procedure or namespace.

3.6.5.1. Local variables

Local variables are defined within a procedure. They are automatically created when set and there is no `global` (Section 3.6.5.2) or `variable` (Section 16.3) command within the procedure that declares them to be global or within a namespace. Local variables exist until the procedure returns or they are explicitly destroyed with the `unset` (Section 3.6.4) command. Procedure parameters are also local variables that are automatically initialized from passed arguments when the procedure is called.

Local variables can also be accessed from procedures called by the one where they are defined. The use of this feature is fully described in Section 14.1.4.

3.6.5.2. Global variables: global

```
global ?VARNAME …?
```

Global variables are variables at the top level of a Tcl interpreter outside of any procedure or namespace context. They exist from the time of definition until they are explicitly destroyed with the `unset` (Section 3.6.4) command. The variables in our interactive examples in the Tcl shell were all created as global variables.

Within a procedure or namespace, global variables have to be either qualified or declared as global. Qualifying a global variable is done by prefixing the variable name with the `::` character sequence which references the global namespace.

```
set globalvar "I am global"
proc demo {} { puts $::globalvar }
demo
→ I am global
```

Alternatively, the variable can be declared to be global with the `global` command.

The command creates local variables in the procedure that are linked to the corresponding global variables of the same name. Unqualified references to that name then dereference the global variable. Thus the above procedure could also be written as

```
proc demo {} {global globalvar; puts $globalvar}
demo
→ I am global
```

Explicit qualification immediately makes it clear at the point of reference that a global variable is being used. On the other hand, declaration using `global` prior to entering a loop that references the variable can be significantly faster.

3.6.5.3. Creation is not definition

We have so far used the words “creation” and “definition” somewhat interchangeably. However, they are not synonyms. Following terminology from the Tcl reference pages, *creation* of a variable refers to creation of the variable **name** and associating it with a scope (local, global, namespace etc.). Commands such as `global`, or `variable` that we see in Chapter 16, perform this function. On the other hand, a variable is *defined* only when a value is assigned to it. The `info exists` command (Section 3.6.7) which checks variable definition illustrates this.

```
global a_global_var           → (empty)
info exists a_global_var      → 0
set a_global_var "some value" → some value
info exists a_global_var      → 1
```

3.6.6. Enumerating variables: info vars|locals|globals

```
info locals ?PATTERN?
info globals ?PATTERN?
info vars ?PATTERN?
```

The `info locals`, `info globals` and `info vars` commands can be used to enumerate local variables within a procedure, global variables, and all variables that are visible in the current scope respectively. The script below illustrates their use.

```
set globalvar "I am global"
proc demo {paramA} {
    set localvar "A local variable"
    puts "locals: [info locals]"
    puts "globals: [info globals]"
    puts "vars: [info vars]"
    global tcl_platform
    puts "vars: [info vars]"
}
demo "A parameter"
→ locals: paramA localvar
  globals: tcl_version tcl_interactive var globalvar bvar nested stdout_chan re...
  vars: paramA localvar
  vars: paramA localvar tcl_platform
```

Some of the variables we see in the output are predefined in Tcl. Others were created as a result of commands executed in our earlier examples.

Notice that `info locals` includes the procedure parameters in its list. Also note the behaviour of `info vars`. Unlike `info globals`, it will only include global variables if they have been brought into the local scope with a `global` declaration. Also, although not shown in our example, `info vars` will list namespace variables that have been brought into the local scope.

In all cases, if *`PATTERN`* is specified, only variables matching the pattern using `string match` rules (Section 4.24) are returned.

```
% info globals tcl_*
→ tcl_version tcl_interactive tcl_patchLevel tcl_platform tcl_library
```

If the pattern includes namespaces, only the last component of the namespace variable is treated as wildcard pattern. The namespace names are treated as literals. So for example,

```
info vars ns::* → (empty)
info vars *::*  → (empty) ❶
```

❶ Returns empty list because the namespace component `*` is treated as a literal namespace and not as a wildcard.

3.6.7. Checking variable existence: info exists

```
info exists VARNAME
```

The `info exists` command checks for the existence of a variable within any scope. It returns `1` if a variable exists and `0` otherwise. Here *exists* means the variable is *defined*, i.e. has been created **and** has an associated value. Notice from the output below that `info exists` follows the same rules regarding qualification and global declarations as any other variable reference.

```
proc demo {} {
    puts "localvar: [info exists localvar]"
    set localvar "I am local"
    puts "localvar: [info exists localvar]"
    puts "globalvar: [info exists globalvar]"
    puts "::globalvar: [info exists ::globalvar]"
    global globalvar
    puts "globalvar: [info exists globalvar]"
}
demo
→ localvar: 0
  localvar: 1
  globalvar: 0
  ::globalvar: 1
  globalvar: 1
```

3.6.8. Array variables

Many languages provide constructs referred to as *maps* or *associative arrays* where values are stored as elements in a collection and referenced using a *key*. In Tcl, *arrays* are similar except that they are actually not a collection of values but rather a *collection of variables*. They are denoted using the special variable syntax

```
ARRAYNAME(KEY)
```

where both the array name and key may be arbitrary strings.

Tcl also has *value* based keyed collections called *dictionaries*. We describe dictionaries, and contrast them with arrays, in Chapter 6.

3.6.8.1. Basic array operations

Because array elements are just variables, they are used in the same fashion. We can access them with the `$` prefix and use any commands like `set`, `append`, `incr` etc. that operate on variables to modify an element.

Let us define an array that maps cities to their population.

```
% set populations(Mumbai) 21673000
→ 21673000
% puts "The population of Mumbai is now $populations(Mumbai)."
→ The population of Mumbai is now 21673000.
% puts "Next year it will be [incr populations(Mumbai) 1000000]"
→ Next year it will be 22673000
```

The key is not restricted to being a literal string.

```
set city "New York"                  → New York
set populations($city) 8260000       → 8260000
proc ukcapital {} {return London}    → (empty)
set populations([ukcapital]) 9760000 → 9760000
```

3.6.8.2. Array defaults: array default

```
array default set ARRAYNAME VALUE
array default unset ARRAYNAME
array default get ARRAYNAME
array default exists ARRAYNAME
```

The `array default` command configures a default value to be returned when an attempt is made to retrieve a key that does not exist in the array. The `array default` command ensemble allows setting, removing, retrieving and checking for existence of a default value for array elements that is returned for keys that are not present in an array.

```
array set fonts {
    code Courier
    headings Helvetica
}                              → (empty)
array default exists fonts     → 0
puts $fonts(body)              Ø can't read "fonts(body)": no such element in array
array default set fonts Arial → (empty)
array default exists fonts     → 1
puts $fonts(body)              → Arial ❶
info exists fonts(body)        → 0 ❷
array default unset fonts      → (empty)
puts $fonts(body)              Ø can't read "fonts(body)": no such element in array
```

❶ Default value returned.
❷ The element is not actually present.

The `array default` command is not available in Tcl 8.6 and earlier. Instead an explicit check has to be made for the existence of an element using `info exists` before retrieving its value.

```
if {[info exists fonts(body)]} {puts $fonts(body)} else {puts "Arial"}
→ Arial
```

3.6.8.3. Checking for arrays: array exists

The `array exists` command returns 1 if a variable is an array, and `0` otherwise.

```
set scalar "some value"  → some value
array exists populations → 1
array exists scalar      → 0
array exists nosuchvar   → 0
```

3.6.8.4. Checking for element existence: info exists, array names

```
info exists ARRAYNAME(KEY)
array names ARRAYNAME ?MODE? ?PATTERN?
```

Since array elements are variables, the `info exists` command, which checks the existence of a variable, can be also used to check for array elements.

```
info exists populations(Mumbai)   → 1
info exists populations(Helsinki) → 0
```

The `array names` command returns the keys[7] in an array.

The command returns all keys in the array or just those matching *`PATTERN`* if specified. If *`MODE`* is unspecified or `-glob`, the pattern matching rules for `string match` (Section 4.24) are used. If *`MODE`* is `-regexp`, *`PATTERN`* is treated as a regular expression (Section 10.1). If *`MODE`* is `-exact`, only the key that exactly matches *`PATTERN`* is returned.

The command returns an empty list if no matching elements are found, or if *`ARRAYNAME`* is not an array variable.

```
array names populations              → London {New York} Mumbai
array names populations M*           → Mumbai
array names populations -regexp o..o → London
```

3.6.8.5. Operating on multiple elements: array set|get|unset

```
array set ARRAYNAME VALUELIST
array get ARRAYNAME ?PATTERN?
array unset ARRAYNAME ?PATTERN?
```

Although individual array elements can be treated like any other variable, it is also possible to operate on multiple elements at a time with commands meant for that purpose.

The `array set` command assigns to multiple elements. *`ARRAYNAME`* is the name of the variable which must be an array if it already exists. *`VALUELIST`* is a list of alternating keys and

[7] The Tcl reference documents uses the nomenclature *names* for keys in an array and *keys* for a dictionary. We stick to using *keys* for both cases.

values. Each value is assigned to the array element identified by the corresponding key. The command does not affect any existing elements whose keys are not present in *VALUELIST*.

```
array set populations {
    Moscow 12200000
    Lagos  17000000
    Mumbai 12500000
}
```

Conversely, `array get` returns multiple elements as a list of alternating keys and values.

If *PATTERN* is not specified, the command returns all elements in the array. Otherwise, only elements whose keys match *PATTERN* using `string match` rules (Section 4.24) are returned.

```
% array get populations
→ Moscow 12200000 Lagos 17000000 London 9760000 {New York} 8260000 Mumbai 12500000
% array get populations M*
→ Moscow 12200000 Mumbai 12500000
```

The elements are returned in an arbitrary order. If a specific order is desired, you can use the `lsort` command (Section 5.21).

```
% lsort -nocase -stride 2 [array get populations]            ❶
→ Lagos 17000000 London 9760000 Moscow 12200000 Mumbai 12500000 {New York} 8260000
% lsort -integer -index 1 -stride 2 [array get populations]  ❷
→ {New York} 8260000 London 9760000 Moscow 12200000 Mumbai 12500000 Lagos 17000000
```

❶ Sort by name
❷ Sort by population

Finally, while `unset` can be used with array elements as with other variables, `array unset` provides a means to unset multiple array elements.

If *PATTERN* is not specified, the entire array is unset. Otherwise, only those elements whose keys match *PATTERN* using `string match` rules (Section 4.24) are unset. The command has no effect if *ARRAYNAME* does not exist or is not an array.

```
array names populations    → Moscow Lagos London {New York} Mumbai
array unset populations L* → (empty)
array names populations    → Moscow {New York} Mumbai
```

Note the difference in the following commands

```
array unset my_array *
array unset my_array
```

The first will remove all elements from the array but the array itself will continue to exist. In the second case, the array variable itself will be unset.

3.6.8.6. Iterating over arrays: array for|startsearch|nextelement|anymore|donesearch

```
array for {KEYVAR VALUEVAR} ARRAYNAME BODY
```

The `array for` command iterates over all elements of an array.

The command executes the script *`BODY`* for every element in the array assigning the key and value of an element to the variables *`KEYVAR`* and *`VALUEVAR`* on each iteration. The command does not guarantee the order in which elements are iterated. It also does not permit elements to be added or removed during the iteration.

```
% array for {city population} populations {
    puts "$city: $population"
}
→ Moscow: 12200000
  New York: 8260000
  Mumbai: 12500000
```

The `array for` command is not supported in Tcl 8.6 or earlier. Use one of the alternatives below if working with those versions.

Alternatively, the `foreach` command (Section 5.8) can be used to iterate over array content using `array names` or `array get`.

```
foreach {city population} [array get populations] {
    puts "The population of $city is $population"
}
→ The population of Moscow is 12200000
  The population of New York is 8260000
  The population of Mumbai is 12500000
```

Or if sorted order is desired,

```
foreach city [lsort -dictionary [array names populations]] {
    puts "The population of $city is $populations($city)"
}
→ The population of Moscow is 12200000
  The population of Mumbai is 12500000
  The population of New York is 8260000
```


The `array for` command is significantly more efficient than `foreach`. However, the latter allows for addition and deletion of elements and is also compatible with older Tcl versions.

There is yet another method for iterating over arrays. Like `array for`, it is efficient in memory usage but is also available in Tcl 8.6 and earlier. It is however slower and less convenient to use.

The first step involves retrieval of a handle to an iterator using `array startsearch`.

```
set iter [array startsearch populations] → s-1-populations
```

The `array anymore` command is used in conjunction with `array nextelement`, which retrieves the next element from the iterator, to loop over all the elements. It returns `1` if there are more elements left and `0` otherwise.

```
while {[array anymore populations $iter]} {
    set city [array nextelement populations $iter]
    puts "The population of $city is $populations($city)"
}
→ The population of Moscow is 12200000
  The population of New York is 8260000
  The population of Mumbai is 12500000
```

Finally, when the iteration has ended, the handle has to be released with `array donesearch`.

```
array donesearch populations $iter → (empty)
```

The iterator should not be used subsequently. Addition or deletion of elements also terminates the iteration in the same manner.

It is possible to have multiple iterators simultaneously active on an array.

3.6.8.7. Array statistics: array size, array statistics

```
array size ARRAYNAME
```

The `array size` command returns the number of elements in an array. In case the specified variable does not exist or is not an array, the command returns 0.

```
array size populations → 3
array size nosuchvar   → 0
```

The `array statistics` command is rarely used in practice and included here only for completeness. It is primarily a tool to diagnose pathological behaviour with very large arrays.

```
% array statistics populations
→ 3 entries in table, 4 buckets
  number of buckets with 0 entries: 2
  number of buckets with 1 entries: 1
...Additional lines omitted...
```

3.6.8.8. Printing an array: parray

```
parray ARRAYNAME ?PATTERN?
```

The `parray` command prints the contents of an array. If *`PATTERN`* is not specified, the command prints on standard output all elements of *`ARRAYNAME`* in alphabetic order of the keys. If *`PATTERN`* is specified, the command only outputs elements whose keys match *`PATTERN`* using the pattern matching rules of `string match` (Section 4.24). This command is primarily intended for interactive use.

```
% parray populations
→ populations(Moscow)   = 12200000
  populations(Mumbai)   = 12500000
  populations(New York) = 8260000
% parray populations N*
→ populations(New York) = 8260000
```

3.6.8.9. More on array keys

There are some additional points to be noted about keys.

Key equality

Keys are strings. Thus the keys `1` and `0x1` point to different array elements though in numeric calculations they represent the same values. Additionally, keys are case sensitive so keys `abc` and `Abc` refer to different array elements.

Multiple dimensions

There is no built-in notion of multidimensional arrays. They are sometimes simulated by concatenating the multiple "indices" using some separator string and using the result as the array key. For example, the results of tennis matches may be stored using keys like `Federer,Nadal`. However, you need to be careful that the separator string itself does not occur in the index values as it would lead to ambiguities. Also remember that `Federer,Nadal` and `Federer, Nadal` (with a space before the `N`) are different keys so even extraneous whitespace will lead to erroneous results if not used consistently. For these reasons, dictionaries (Chapter 6) are preferable for such structures.

Empty strings as keys

As a piece of trivia, note that empty strings are acceptable for both the array and the key.

```
set (key) "Array name is empty"      → Array name is empty
set arr() "Array key is empty"       → Array key is empty
set () "Both name and key are empty" → Both name and key are empty
```

Keys containing whitespace

Earlier we assigned an element for the key `United States` via a variable reference. However, direct assignment with a key containing whitespace leads to errors.

```
% set populations(Hong Kong) 7300000
Ø wrong # args: should be "set varName ?newValue?"
% set populations("Hong Kong") 7300000
Ø wrong # args: should be "set varName ?newValue?"
```

The correct way to set the variable is either of

```
set populations(Hong\ Kong) 7300000  → 7300000
set "populations(Hong Kong)" 7300000 → 7300000
```

On the other hand, when referencing the variable we can use a braced variable reference.

```
puts $populations(Hong Kong)   → 7300000
puts ${populations(Hong Kong)} → 7300000
```

3.6.9. Constant variables: const

```
const VARNAME VALUE
info constant VARNAME
info consts ?PATTERN?
```

The `const` command creates and initializes a variable which cannot be subsequently modified. If the variable already existed, it must have been marked as const.

Any attempt to modify or unset the variable will fail. The variable ceases to exist only when its namespace is deleted or the containing procedure exits.

```
const KONST 42          → (empty)
set KONST 43            Ø can't set "KONST": variable is a constant
unset KONST             Ø can't unset "KONST": variable is a constant
unset -nocomplain KONST → (empty)
info exists KONST       → 1
```

The `info constant` command returns `1` if a variable is marked const and `0` otherwise.

```
info constant KONST → 1
```

The `info consts` command returns the list of constant variables. If *PATTERN* is not specified, the command returns the names of constant variables in the current scope. Otherwise, it returns the names matching *PATTERN* using `string match` rules (Section 4.24).

3.6.10. Predefined variables

Tcl predefines a number of global variables such as `tcl_platform`, `tcl_version` etc. You can get a complete list from the `info globals` command in the Tcl shell. We will describe these variables elsewhere in the sections related to their use.

3.7. Conditional execution: if

```
if EXPR then BODY ?elseif EXPR then BODY …? ?else  BODY?
```

The `if` command conditionally evaluates scripts based on the boolean truth values of expressions. Each expression *`EXPR`* is evaluated in the same manner as the `expr` command (Section 7.2.2) and is expected to yield a boolean. The *`BODY`* script corresponding the first *`EXPR`* that evaluates to true is executed. If none of the expressions evaluate to a boolean true value, the *`BODY`* script associated with the `else` clause is evaluated, if present.

```
if {$i > 0} {
    puts "$i is positive"
} elseif {$i < 0} {
    puts "$i is negative"
} else {
    puts "$i is zero"
}
```

The `elseif` and `else` clauses are optional. Multiple `elseif` clauses are permitted. The keywords `then` and `else` are also optional. By convention, the `then` keyword is omitted while `else` is explicitly specified as above.

The result of the `if` command is the result of the evaluated script or an empty string if no expressions yielded true and no `else` clause was present.

```
% set x 2 ; set y 1
→ 1
% set x [if {$x > $y} {set x} else {set y}]
→ 2
```

Coming from other languages, you may try to write your `if`, `while` and other compound statements as follows, placing the braces on a separate line:

```
if {$x > $y}
{
    …do something…
}
else
{
    …do something else…
}
```

This will raise an error. Remember `if` is a command like any other; it is **not** a special keyword with special syntax. Its expressions and script bodies are just arguments as for any other command and the usual syntax rules apply. In the above example, Tcl will see the first line as a complete command with two words and invoke `if` with a single argument, the braced expression. The `if` command will then raise an error as it expects at least two arguments.

3.8. Conditional execution: switch

```
switch ?OPTIONS? ?--? VALUE LIST
switch ?OPTIONS? ?--? VALUE PATTERN BODY ?PATTERN BODY …?
```

The `switch` command is an alternate means for conditional evaluation of scripts. It matches a value against multiple patterns and evaluates a script based on the first matched pattern. It has two syntactic forms. In the first form, *`LIST`* is a list of *`PATTERN`* and *`BODY`* elements specified as a single argument. In the second form, each pair is separately specified.

The command compares the *`VALUE`* argument against each *`PATTERN`* in turn and evaluates the *`BODY`* argument corresponding to the first pattern that matches. If the **last** pattern is `default`, it is considered a match for all values and the corresponding *`BODY`* is executed if no previous pattern matched. Any *`BODY`* element may be specified as the `-` character, in which case the *`BODY`* corresponding to the following pattern is executed.

The optional `--` character sequence is used to separate options from the *`VALUE`* argument in case of any ambiguity arising from the first character of *`VALUE`* being `-`.

An example using the first form of the command:

```
switch $image_format {
    png { save_as_png $image }
    jpg -
    jpeg { save_as_jpeg $image }
    gif { save_as_gif $image }
    default {
        error "Unsupported image type $image_format"
    }
}
```

The same example using the second form of the command would read as

```
switch $image_format png {
    save_as_png $image
} jpg - jpeg {
    save_as_jpeg $image
} gif {
    save_as_gif $image
} default {
        error "Unsupported image type $image_format"
}
```

The fundamental difference between the two forms is that in the second form the patterns being compared undergo substitution (Section 3.2) while those in the first form do not because they are within a braced block. This makes the first form suitable for matching against literal patterns while the second form is more convenient when the patterns are not literals but passed through variables.

The `switch` command returns the result of the evaluated script or the empty string if no pattern matched.

The options shown in Table 3.3 control the type of matching performed.

Table 3.3. Matching options for switch

Option	Description
`-exact`	Exact string matching. This is the default.
`-glob`	Matched as per `string match` rules (Section 4.24).
`-indexvar`	May only be used in conjunction with `-regexp`. Described later.
`-matchvar`	May only be used in conjunction with `-regexp`. Described later.
`-nocase`	Modifies the matching to be case-insensitive. By default matching is case-sensitive.
`-regexp`	Matched as a regular expression (Chapter 10).

Below is an example of using switch with glob patterns. Here we are using `string cat` (Section 4.12) (Section 4.12) as an identity function that simply returns its argument. Note the use of the return value from the `switch` command.

```
set url "https://www.example.com"
set port [switch -glob -nocase -- $url {
    http://*  { string cat 80 }
    https://* { string cat 443 }
    ftp://*   { string cat 21 }
    default   { error "Unknown URL type" }
}]
→ 443
```

In the case of regular expression matching, the options `-matchvar` and `-indexvar` may be specified. The `-matchvar` option takes an additional argument that is the name of the variable in which to store the matched substrings. The content of this variable will be a list whose first element is the entire substring of *`VALUE`* that matched the regular expression pattern. The remaining elements of the list contain the substrings matched by the capturing parenthesis (Section 10.1.10.1) in the expression, if any.

```
proc connect_url {url} {
    switch -regexp -nocase -matchvar connection -- $url {
        "http://([-_%:.[:alnum:]]*)" {
            puts "Connecting to [lindex $connection 1] on port 80"
        }
        "https://([-_%:.[:alnum:]]*)" {
            puts "Connecting to [lindex $connection 1] on port 443"
        }
    }
}
connect_url http://www.example.com
→ Connecting to www.example.com on port 80
```

The `-indexvar` option is similar except that instead of a list of matched substrings, the variable will contain a list of pairs containing the start and end indices of the substrings.

Remember that just as in the `regexp` (Section 10.1) command, a regexp match succeeds if the pattern matches any **substring**, not necessarily the entire string.

```
% connect_url "Please connect to http://www.example.com"
→ Connecting to www.example.com on port 80
```

If the desired behaviour is to match the entire string, use the `^` and `$` anchors. See Chapter 10 for details.

3.9. Looping on a condition: while

```
while EXPR BODY
```

The `while` command executes a script as long as a given expression is true. The argument *`EXPR`* is evaluated as an expression (Section 7.2.2). If the result is a boolean true value, the script argument *`BODY`* is executed. This process is repeated until *`EXPR`* evaluates to a false value. The command always return the empty string as its result.

An example of some rather sophisticated computation using `while`:

```
proc sum {n} {
    set sum 0
    while {$n > 0} {
        incr sum $n
        incr n -1
    }
    return $sum
}
sum 3
→ 6
```

The *`EXPR`* argument should almost always be enclosed in braces. Otherwise the parser will substitute the variable values before passing them to the `while` command. The result may be an error or worse. For example, suppose the above loop were written as

```
while $n {
    ….
}
```

Tcl would replace `$n` with its value, say `3`, and that would be the argument seen by the `while` command and the expression evaluated on every iteration. Since `3` always evaluates to a boolean `true`, the loop would run forever.

The `break` (Section 3.11) and `continue` (Section 3.12) loop control commands may be used within *`BODY`* to terminate the loop or to skip iterations.

3.10. Looping over values: for

```
for INIT EXPR NEXT BODY
```

The other generic looping command is the `for` command. The command starts off by executing the script *`INIT`*. It then evaluates *`EXPR`* as an expression. If the result is a boolean true value, the command executes the script *`BODY`*, followed by the script *`NEXT`*. It then repeats this sequence as long as the *`EXPR`* expression evaluates to true. Note that *`INIT`* and *`NEXT`* are also scripts like *`BODY`*, containing zero or more commands. The result of the command is always the empty string.

As always, the `break` (Section 3.11) and `continue` (Section 3.12) commands can be used to control the loop execution. In the case of `continue`, the commands in *`BODY`* after the `continue` will be skipped but the *`NEXT`* script is still executed.

The `while` loop from the previous section could also be written with `for`:

```
proc sum n {
    for {set sum 0} {$n > 0} {incr n -1} {
        incr sum $n
    }
    return $sum
}
```

Tcl has specialized commands `foreach` (Section 5.8) and `lmap` (Section 5.19) for iterating over lists and `dict for` (Section 6.15) and `dict map` (Section 6.16) for iterating over dictionaries. They are discussed in the related chapters.

If you are not satisfied with the variety of looping and control statements in Tcl, it is almost trivial to write your own constructs. See Section 15.7.

3.11. Terminating loops: break

```
break
```

The `break` command is used for prematurely terminating a loop.

Here we copy files to a floppy drive until there is insufficient space. Yes, I'm dating myself and no, not a robust algorithm, does not consider disk allocation quanta and so on.

```
foreach file [glob -nocomplain *] {
    set size [file size $file]
    if {$size > $floppy_size} {
        break
    }
    file copy $file $floppy_drive
    set floppy_size [expr {$floppy_size - $size}]
}
```

3.12. Skipping loops: continue

```
continue
```

The `continue` command is used to abort the **current** iteration of a loop. Let us rewrite our previous example to be slightly smarter.

```
foreach file [glob -nocomplain *] {
    set size [file size $file]
    if {$size > $floppy_size} {
        continue
    }
    file copy $file $floppy_drive
    set floppy_size [expr {$floppy_size - $size}]
}
```

Instead of breaking out of the loop, we now skip the file and move on to the next.

3.13. Evaluating strings: eval

```
eval ARG ?ARG …?
```

One of the major features of dynamic languages is the ability to execute scripts constructed "on-the-fly" in the course of a program's execution. This ability is useful in diverse situations like application macros, text transforms, domain-specific languages and metaprogramming, examples of which we will see in later chapters.

While several commands in Tcl are applicable to the above, the fundamental one is `eval`. The command accepts one or more arguments, concatenates these with a space separator in the same manner as the `concat` command (Section 5.18). It then executes the result as a standard Tcl script.

The result of the command is the result of the script execution. In its simplest form

```
eval {puts foo} → foo
```

the command executes the script `puts foo` and is effectively no different than

```
puts foo → foo
```

Let us look at an example that illustrates the difference. We will define a variable `bar` with a value `hello` and a second variable `foo` which references it.

```
set bar "hello" → hello
set foo {$bar}  → $bar ❶
```

❶ Note the braces cause `$bar` to be treated as a literal string.

Now compare the following commands.

```
puts $foo      → $bar
eval puts $foo → hello
```

By now you should have understood why the `puts $foo` command on the first line prints `$bar` — Tcl does not reparse the words comprising a command after any substitutions are made (Section 3.2). The `eval` on the other hand prints `hello`. Let us look at this `eval` statement step by step:

- When it is parsed by Tcl, the `$foo` variable reference is replaced by its value `$bar`.
- The `eval` command receives two arguments, the strings `puts` and `$bar`.
- It concatenates its arguments to form the string `puts $bar`.
- It then evaluates this string as a Tcl script. As per the usual rules of substitution, the string is broken up into words with `$bar` replaced by the variable value `hello`.
- The command `puts` is then invoked with the argument `hello` and does its thing.

3.13.1. Double substitutions in eval

We see in the above example that it appears that the variable reference `$foo` undergoes double substitution in the command `eval puts $foo`, first to `$bar` and then to `hello`. Note this does not contradict what we stated earlier about the Tcl parser not reparsing substituted values. Here it is the `eval` command that is invoking the Tcl parser a second time. Remember we said Tcl commands can do whatever they want with their arguments? Here `eval` chooses to treat its (concatenated) arguments as a Tcl program to be parsed and executed.

Contrast the previous example with braced arguments to `eval`:

```
eval {puts $foo} → $bar
eval puts {$foo} → $bar
```

In both these cases, the braces prevent the **initial** round of substitutions. The `eval` command still does its concatenation and substitution, but because it is now passed `$foo` as its argument, and not the value of `foo`, a single round of substitution results.

It needs to be emphasized that `eval` executes a **script** and not a **command**. Thus both the following lines are parsed as a script with two commands and not as a single command `puts` with four arguments `foo`, `;`, `puts` and `bar`.

```
% eval {puts foo ; puts bar}
→ foo
  bar
% eval "puts foo" ";" puts bar
→ foo
  bar
```

Issues around double substitutions and quoting come up with several other commands and can be confusing so we will take up a couple of additional examples.

We first define some variables used in the examples.

```
set cmd "string length" → string length
set arg "foo bar"       → foo bar
```

Now compare the results of the various commands below.

```
$cmd $arg             Ø invalid command name "string length" ❶
eval {$cmd $arg}      Ø invalid command name "string length" ❷
eval $cmd $arg        Ø wrong # args: should be "string length string" ❸
eval $cmd {$arg}      → 7
eval $cmd [list $arg] → 7
```

❶ Fails because `string length` is parsed as a *single* word and there is no command of that name.
❷ Fails for the same reason as above.
❸ Fails because double substitution causes `foo bar` to be treated as two arguments `foo` and `bar`, whereas `string length` expects a single argument.

Make a note of the last two forms. In the first of these, the word `$cmd` is substituted as `string length` while the braces around the argument prevent the first round of variable substitution. What `eval` sees are two arguments `string length` and `$arg`. These are concatenated into a single string `string length $arg` and run through the parser which now breaks them up into three words `string`, `length` and `foo bar` (after substitution of `$arg`) which are evaluated as a command.

The second form using the list command works similarly except that instead of wrapping in braces, it wraps `$arg` as a one-element list achieving essentially the same effect (Section 5.1).

In our example, the two are effectively the same because the `eval` command does not change the variable context and none of the arguments have side effects. However, as we will see in later sections, the difference is important for commands like `uplevel` (Section 14.1.5) that can execute in different variable contexts.

In earlier versions of Tcl, a common use for `eval` was to expand a list value into its constituent elements. We saw one example above where `$cmd` was expanded into two words `string` and `length`. In modern versions of Tcl (8.5 and later) the recommended method is to use the argument expansion syntax (Section 3.4) instead.

```
{*}$cmd $arg → 7
```

In general, use of `{*}` mitigates the possibility of inadvertent double substitution. See https://wiki.tcl-lang.org/page/eval for more on `eval` and double substitution.

3.14. Evaluating file content: source

```
source ?-encoding ENCODING? PATH
```

We have already seen in Section 2.2.2.2 how a Tcl program stored in a file can be executed by passing the file name as a command line argument to the `tclsh` or `wish` applications. The `source` command is another means of evaluating the contents of a file as a Tcl script.

The command reads the file identified by *`PATH`* and evaluates its content as a Tcl script in a manner similar to `eval` (Section 3.13). If *`PATH`* is a relative path, it is relative to the **current** working directory, **not** relative to the file containing the `source` command.

If the file content is not in ASCII or UTF-8 encoding (Section 9.1), the `-encoding` option should specify the correct encoding.

In Tcl 8, sourced content is assumed to be in the system encoding, not UTF-8. In the author's opinion, for compatibility reasons scripts should stick to ASCII encoding which is valid for UTF-8 and most other encodings. Any non-ASCII characters can be represented with Unicode backslash sequences (Section 3.2.1).

The presence of a Ctrl-Z character in the file content is treated as the end of the file by the `source` command. Any data beyond the Ctrl-Z character is ignored. This feature is sometimes used to store binary data beyond the end of the Tcl script. The script can use the `info script` command (Section 3.14.1) to identify its containing file, read it in and then locate the binary data by searching for the first Ctrl-Z character. This is often more convenient than having to distribute a separate file containing the data.

The result of the `source` command is that of the last command executed in the script. A `return` command (Section 15.3) within the script will cause the rest of the commands in the script to be skipped with the argument to `return` returned as the result.

It is perfectly legal to source a file multiple times. This is particularly useful during interactive development where you might edit the source code to fix bugs or add features and then re-source the file into the interpreter.

Most large applications and packages are structured as a single "main" script with supporting data and procedure definitions stored in other files. Running the application or loading a package involves executing this script which in turn uses `source` to pull in the other files.

3.14.1. Retrieving script paths: info script

```
info script ?SCRIPTPATH?
```

It is often useful for a script sourcing additional support scripts to know its own path so that it can locate the other scripts it needs to source. The `info script` command provides this information.

In the usual case, where the *`SCRIPTPATH`* argument is not specified, the command returns the full path of the innermost file being sourced. For example, if file `a.tcl` is being sourced and it

in turn sources `b.tcl` which in turn sources `c.tcl`, the result of the command **while** `c.tcl` **is being sourced** will be the full path to `c.tcl`.

The command can be used in the main script in a fashion similar to the code fragment below.

```
namespace eval myapp {
    # Remember the directory we are located in.
    variable script_dir [file dirname [info script]]
}
source [file join $::myapp::script_dir a.tcl]
source [file join $::myapp::script_dir b.tcl]
…
```

The command returns an empty string if no file is being sourced.

The following procedure to return the directory where the file containing the procedure is located will **not** work as you might expect.

```
proc get_my_dir {} { return [file dirname [info script]] }
```

If the procedure is called **after** the file has been sourced, `info script` returns the empty string which is not what you would want. The `info script` command must be executed while the file is being sourced.

When the command is supplied the *`SCRIPTPATH`* argument, further calls to the `info script` command will return *`SCRIPTPATH`* instead of the real file name **for the duration of the current source command**.

Dual mode scripts

In many instances, a Tcl script may run as a main application or be loaded as a library module. For example, a script may run as a command line Web client when invoked from the command line or simply provide a library for retrieving Web pages when loaded as a package into an application. The `info script` command can be used to distinguish the two cases:

```
if {[info exists ::argv0] &&
    [file dirname [file normalize [info script]/…]] eq [file dirname [file
 normalize $argv0/…]]} {
    … Script file was passed as an argument on the command line …
    … Parse command line options and retrieve web pages …
} else {
    … Script was sourced as a library
}
```

The only points to be noted are the need to normalize before comparing the command line argument `argv0` which contains the path of the script invoked from the command line. This normalization takes care of relative and absolute path differences as well links. See Section 12.1.6 regarding the above normalization pattern.

3.15. Introspection

> Three things extremely hard: steel, a diamond, and to know one's self.
>
> — Benjamin Franklin

Well, the great man had clearly not heard of Tcl. Tcl offers deep and comprehensive introspection capabilities into almost every aspect of its runtime. Introspection is useful in all kinds of situations ranging from metaprogramming, runtime debugging and tracing, construction of dynamic object systems and more. It is even useful in interactive development. For example, what arguments does our `demo2` procedure take?

```
proc demo2 {x y} {} → (empty)
info args demo2     → x y
```

In most cases this information is available through the `info` command. We have already seen a few examples such as `info procs` and `info globals`. We will describe these introspection capabilities in detail in the relevant sections throughout this book.

3.16. Getting error information

Tcl has powerful mechanisms for dealing with errors and exceptions that we describe in Chapter 15. Here we only mention a couple of points that are useful to know when starting with Tcl in interactive mode.

Most Tcl commands will print an informational message that identifies the cause of the error.

```
% binary decode hex
Ø wrong # args: should be "binary decode hex ?options? data"
% string size "foo"
Ø unknown or ambiguous subcommand "size": must be cat, compare, equal, first,
   ↳ index, insert, is, last, length, map, match, range, repeat, replace, reverse,
   ↳ tolower, totitle, toupper, trim, trimleft, trimright, wordend, or wordstart
```

In addition, if an error occurs in a nested procedure call, you can examine the global variable `errorInfo` for the call stack at the point the error occurred.

```
% proc demo args { demo2 {*}$args }
% demo A B C
Ø wrong # args: should be "demo2 x y"
% puts $errorInfo
→ wrong # args: should be "demo2 x y"
      while executing
  "demo2 {*}$args "
      (procedure "demo" line 1)
      invoked from within
  "demo A B C"
```

This can be very useful in diagnosing the root cause of an error.

If you are using an enhanced Tcl console like `tkcon`, error messages are highlighted and clicking on them with the mouse will display the error stack in a popup window.

3.17. The EIAS principle

Having looked at the basics of the language, let us touch upon a core philosophy on which Tcl is based — EIAS (*Everything Is A String*).

- Every **value** has a string representation. A "string" as we see in the next chapter, is a finite sequence of characters supporting operations that return its length, indexing and so on. This also means that every value is automatically serializable.
- Every value that produces the same string representation must be treated by every command in exactly the same way no matter how those values were constructed. For example, a value with the string representation `100` may arise as the concatenation of the strings `10` and `0` or as the result of squaring the numeric value `10`. The result of both operations must be treated by all commands in the same manner. An arithmetic operation requiring numeric operands cannot accept the second value and reject the first.
- Arguments to procedures, values stored in variables, etc. are **conceptually** passed and stored as strings though the **implementation** may not do so for reasons of efficiency.
- Because of the above, there is no need for mechanisms such as templates or generics because all values are treated uniformly. Your hash table can contain any value without needing "type-specific" versions.
- Although everything is a string to Tcl, commands are free to operate only on a subset of values in the string universe. The arithmetic operations will only operate on the subset of values that represent numbers.
- A program element **can** also be a string. That includes, for example, procedure bodies. You can dynamically construct procedure definitions as strings and invoke them. However, not all program elements are strings. Namespaces, interpreters are not themselves strings though they have names that are. This does not violate EIAS because they are not **values**. Thus EIAS is perhaps better named as EVIAS (Every Value Is A String).

Having looked what EIAS **is**, let us dispel some myths by stating what it **isn't**.

- EIAS does not mean Tcl has no facilities for numerics, structured data etc.
- EIAS does not mean that all data is internally stored in string form.
- EIAS does not mean operations on numbers and structures entail conversion back and forth from string forms on every use.

Much of Tcl's malleability and ease of programming comes from this uniform treatment of values prescribed by EIAS.

4

Strings

> The universe is a symphony of vibrating strings.
>
> — Michio Kaku

Just as in the universe, strings are the fundamental type in Tcl. We referenced this earlier. In this chapter, we describe all the facilities for manipulating them in Tcl.

4.1. What is a string

For the most part, you can consider a string in Tcl to be a sequence of characters, in line with the common usage of the word. More accurately though, a *Tcl string* is a sequence of *Unicode code points*. This section aims to condense hundreds of pages of the Unicode standard into a few paragraphs, glossing over pretty much everything!

You can safely skip this section on a first reading, continuing to treat a string as a sequence of characters, and return to it at a later point.

4.1.1. Tcl and Unicode

The Unicode[1] standard defines a set of abstract characters that encompass those used in almost all languages, past and present, on the planet[2]. The term *character* is itself used with slightly differing meanings even within the standard. The definition we go with here is

> *Characters are the abstract representations of the smallest components of written language that have semantic value.*

Amongst other attributes, the standard assigns each character a name and a numerical value in the range 0 to 0x10ffff. This numerical value is a *Unicode code point*. It is referenced in text either by its name or its numeric value expressed in the form `U+HHHHHH` where `HHHHHH` is a sequence of up to 6 hexadecimal digits.

A Tcl string then is a sequence of Unicode code points. This differs from a sequence of characters as humans typically perceive them because the mapping between Unicode abstract characters (or code points) and human-perceived characters is not one-to-one. In particular, the same human-perceived character may be represented by more than one Unicode code point sequence.

[1] https://home.unicode.org/
[2] Efforts are underway to include languages from other planets as well, Klingon in particular.

An example may add some clarity. Both `e` and `é` are recognized as characters by humans. They are also defined as Unicode abstract characters where `e` is an abstract character with the name `LATIN SMALL LETTER E` mapped to the code point U+0065, and `é` is named `LATIN SMALL LETTER E WITH ACUTE` mapped to the code point U+00E9. What might not be obvious is that the diacritical mark `´` is also an abstract character with the name `COMBINING ACUTE ACCENT` mapped to the code point U+0301 even though humans may not recognize it as a character. Such *combining characters* are meant to be used as modifiers for preceding Unicode code points.

The key point is that é may be represented by the single Unicode code point `U+00E9` or the sequence `U+0065 U+0301`.

From the human perspective, the two are semantically the same and look identical when displayed. We can see this in the following output[3] which uses the backslash sequences for Unicode from Table 3.1.

```
puts \u00e9       → é
puts \u0065\u0301 → é
```

From Tcl's perspective however, the two are different strings.

```
string length \u00e9              → 1
string length \u0065\u0301        → 2
string equal \u00e9 \u0065\u0301  → 0
```

This incongruity is not unique to Tcl but is common to many other programming languages.

Despite all the above, in the interest of readability, this book mostly uses the term character in lieu of Unicode code point.

Tcl 8.6 only supports code points in the range `U+0000 - U+FFFF`.

4.2. String indices

Commands that manipulate strings use *indices* to indicate character positions within the string. Indices are 0-based, meaning the index 0 refers to the first character, 1 to the second, and so on. They take one of the following forms which allow for simple arithmetic on indices.

```
INTEGER[(+|-)INTEGER]
end[(+|-)INTEGER]
```

INTEGER may either be an integer literal or a reference to a variable containing an integer. The word `end` refers to either the last character of a string or the position after the last character, **depending on the command**.

[3] Correct display will depend on the ability of your terminal to handle Unicode.

4.3. String literals

We have already seen the use of string literals, with and without string interpolation (Section 3.3.1, Section 3.3.2), and the use of backslash sequences (Section 3.2.1) for non-ASCII and control characters. Refer back to those sections for details.

4.4. Counting characters: string length

```
string length STRING
```

The `string length` command returns the number of characters in a string.

```
string length "Hello, World!" → 13
```

4.5. Retrieving a character by position: string index

```
string index STRING INDEX
```

The command `string index` returns the character at position *`INDEX`* in a string. The index `end` refers to the last character in the string.

```
set pos 4                          → 4
string index "Hello, World!" $pos  → o
string index "Hello, World!" end   → !
string index "Hello, World!" $pos+3 → W
string index "Hello, World!" end-5  → W
```

4.6. Retrieving substring ranges: string range

```
string range STRING FIRST LAST
```

The command `string range` returns the string composed of all characters between two indices *`FIRST`* and *`LAST`* (inclusive) in *`STRING`*. The `end` index refers to the last character.

```
string range "Hello, World!" 0 4        → Hello
string range "Hello, World!" $pos+2 end →  World!
```


The `string range` command extracts substrings based on their position. For extracting substrings based on content, use regular expressions (Section 10.1).

4.7. Inserting characters: string insert

```
string insert STRING INDEX INSERTION
```

The `string insert` command returns the string created by inserting the string *INSERTION* at the position specified by *INDEX* in the string *STRING*. Unlike for `string index` or `string range`, the `end` index refers to the position **after** the last character.

```
string insert abc 0 XYZ     → XYZabc
string insert abc end-1 XYZ → abXYZc
string insert abc end XYZ   → abcXYZ
```

The `string insert` command is not available in Tcl 8.6 and earlier.

4.8. Appending characters: append

```
append VAR ?STRING …?
```

The `append` command appends zero or more strings passed as the *`STRING`* arguments to the content of the variable *`VAR`* and stores the result back in the variable. For efficiency reasons, the `append` command is generally preferred to `string insert` or string interpolation. The command defines the variable if it does not already exist.

```
append newvar "Hello"      → Hello ❶
set who "World"            → World
append newvar " " $who "!" → Hello World! ❷
```

❶ Creates the variable `newvar` if it does not exist
❷ `append` can take multiple arguments

A dollar-sign is not prefixed when passing the variable `newvar` to this command because the command expects the *name* of a variable, rather than the *value* contained in it.

4.9. Replace or delete ranges: string replace

```
string replace STRING FIRST LAST ?REPLACEMENT?
```

The command `string replace` returns the result of replacing the characters between indices *`FIRST`* to *`LAST`*, inclusive, with the string *`REPLACEMENT`*. The `end` index refers to the last character in the string. If *`REPLACEMENT`* is unspecified or the empty string, `string replace` functions as a delete command. There is no dedicated command in Tcl for substring deletion.

```
% string replace "Hello, World!" 0 4 Goodbye
→ Goodbye, World!
% string replace "Hello, World!" 5 end-1
→ Hello!
```


When the substring to be deleted is the leading or trailing substring, use the `string range` (Section 4.6) command instead.

4.10. Replace or delete substrings: string map

```
string map ?-nocase? MAPPING STRING
```

The `string map` command replaces all occurrences of one or more substrings. *`MAPPING`* is a list with alternating elements being the strings to be replaced and their replacement. The command replaces **all** occurences of the former in *`STRING`* and returns the result.

```
string map {ab Q cd XYZ} abacdabccd      → QaXYZQcXYZ
string map {rma {o} o {}} "Hello Norma!" → Hell No!
```

This last example illustrates that the target string is iterated over exactly once. There is no rescanning of replacements against the mapping list. So after `rma` is replaced with `o`, the `o` itself does not get replaced with an empty string.

A related point is that the order of strings in the mapping list is crucial if one match string is a prefix of another. The latter should appear **first** else it will never match.

```
string map {bc XX bcd YYY} abcdabcbdabc → aXXdaXXbdaXX
string map {bcd YYY bc XX} abcdabcbdabc → aYYYaXXbdaXX
```

The string comparisons are case-sensitive unless the `-nocase` option is specified.

```
string map {bC XY} abcdabcbdabc          → abcdabcbdabc
string map -nocase {bC XY} abcdabcbdabc → aXYdaXYbdaXY
```


A more flexible but less efficient command, `regsub`, that has similar functionality based on regular expressions is described in Section 10.2.

The `string map` command can also be used to delete substrings by specifying the empty string as the replacement value.

```
% string map {a {} e {} i {} o {} u {}} "a quick brown fox jumps over a lazy dog."
→  qck brwn fx jmps vr  lzy dg.
```

4.11. Trimming character sets: string trim|trimleft|trimright

```
string OP STRING ?CHARS?
```

The `string trimleft`, `string trimright` and `string trim` commands delete leading and/or trailing characters belonging to a set. *`OP`* may be one of `trimleft`, `trimright` or `trim`. The commands delete all occurrences of any character present in the string *`CHARS`* from the start, end or both sides of *`STRING`* respectively. *`CHARS`* defaults to all whitespace characters.

```
% set s "\t   Hello, World   \n"
→            Hello, World

% string trimleft $s
→ Hello, World

% string trimright $s
→            Hello, World
% string trim $s
→ Hello, World
```

A different set of characters can be trimmed by passing `CHARS`.

```
string trimleft "Hello, World!" "lHe!" → o, World!
```

4.12. Concatenating strings: string cat

```
string cat ?STRING …?
```

The `string cat` command is an alternative to using string interpolation in literals for concatenating strings. The command returns the concatenation of all its arguments.

```
set greeting Hello; string cat $greeting " World!" → Hello World!
```

While string interpolation serves a similar purpose, `string cat` is sometimes more convenient in cases where some of the strings being concatenated need to undergo variable and command substitution and others do not.

The command is also useful when we need to return a concatenation of one or more strings from a script body. Here is an example using the `lmap` command to construct a list.

```
set names [list Kwamina Alexander]
set ages [list 6 8]
lmap name $names age $ages {
    string cat $name " - " $age " " years
}
→ {Kwamina - 6 years} {Alexander - 8 years}
```

The `lmap` command (Section 5.19) constructs a list whose elements are the result of successive evaluation of a script. In this simple example, we want to construct a new list whose elements are formed from the corresponding elements of `names` and `ages` list. Since `lmap` elements are comprised of results of the last command executed within the script on each iteration, we cannot directly use string interpolation to construct the elements. The braces in the output above come from string representation of the constructed list (Section 5.2).

4.13. Joining strings with separators: join

```
join LIST ?SEPARATOR?
```

The `join` command concatenates the elements of a list with a separator string, defaulting to a space character, between consecutive elements.

```
set quote [list "I came" "I saw" "I conquered"] → {I came} {I saw} {I conquered}
join $quote                                     → I came I saw I conquered
join $quote ", "                                → I came, I saw, I conquered
join $quote ""                                  → I cameI sawI conquered
```

4.14. Repeating strings: string repeat

```
string repeat STRING COUNT
```

The `string repeat` command returns the *`STRING`* repeated *`COUNT`* times. So to underline a title, for example,

```
% set title "Underlined title"
→ Underlined title
% puts "$title\n[string repeat - [string length $title]]"
→ Underlined title
  ----------------
```

4.15. Changing case: string tolower | toupper | totitle

```
string tolower STRING ?FIRST? ?LAST?
string toupper STRING ?FIRST? ?LAST?
string totitle STRING ?FIRST? ?LAST?
```

The commands, `string tolower`, `string toupper` and `string totitle` change the character case of a string. The meaning of the first two should be obvious. The last capitalizes the first letter in the string and changes all remaining letters to lower case.

```
string tolower "Hello, World!" → hello, world!
string toupper "Hello, World!" → HELLO, WORLD!
string totitle "hELLO, WORLD!" → Hello, world!
```

The optional *`FIRST`* and *`LAST`* indices demarcate the substring range to be modified.

```
string tolower "Hello, World!" 0 4       → hello, World!
string tolower "Hello, World!" 7         → Hello, world!
string toupper "Hello, World!" end-5 end → Hello, WORLD!
string totitle "Hello, World!" 1 end     → HEllo, world!
```

4.16. Reversing a string: string reverse

```
string reverse STRING
```

The `string reverse` command reverses the order of characters in a string. For example, Napolean's lament

```
string reverse "able was I ere I saw elba" → able was I ere I saw elba
```

Hmm... probably not a well chosen example!

4.17. Searching for substrings: string first|last

```
string first NEEDLE HAYSTACK START
string last NEEDLE HAYSTACK START
```

The commands `string first` and `string last` return the location of a substring. The commands look for the first and last occurence of *`NEEDLE`* in *`HAYSTACK`* respectively, returning the index of the first character of the occurence if found and `-1` otherwise.

```
string first "da" "Madam, I'm Adam" → 2
string last "da" "Madam, I'm Adam"  → 12
```

The optional parameter *`START`* designates the starting point of the search. Note the semantics in the case of `string last` which is in essence a search starting from the end.

```
string first "da" "Madam, I'm Adam" 3    → 12
string last "da" "Madam, I'm Adam" end-5 → 2
```


Regular expressions, covered in Chapter 10, provides additional powerful and flexible facilities for search.

4.18. Searching for word boundaries

```
tcl_endOfWord STRING START
tcl_startOfNextWord STRING START
tcl_startOfPreviousWord STRING START
tcl_endOfWord STRING START
tcl_wordBreakBefore STRING START
```

Tcl provides a few commands to locate the word boundaries in a string. Recognition of word boundaries is based on the value of two variables: `tcl_wordchars` and `tcl_nonwordchars`. The content of these two variables should be regular expressions that match word characters and non-word characters respectively. By default, they are set to `\w` and `\W` (Table 10.3).

- `tcl_endOfWord` command returns the index of the first non-word character following the first word character after *`START`*.
- `tcl_startOfNextWord` returns the index of the first word character following the first non-word character after *`START`*.
- `tcl_startOfPreviousWord` returns the index of the first word character following the first non-word character occurring before *`START`*.
- `tcl_wordBreakAfter` returns the index of the first word break following *`START`*. A word break is a pair of characters comprising one word and one non-word character in any order. The index returned by the command is that of the second character in the pair.
- `tcl_wordBreakBefore` returns the index of the first word break preceding *`START`*. A word break is pair of characters comprising one word and one non-word character in any order. The index returned by the command is that of the second character in the pair.

All commands return `-1` on failure to locate the relevant boundary.

```
tcl_endOfWord "first second third" 0           → 5
tcl_startOfNextWord "first second third" 0     → 6
tcl_wordBreakAfter "first second third" 0      → 5
tcl_startOfPreviousWord "first second third" 6 → 0
tcl_wordBreakBefore "first second third" 6     → 6
```

4.19. Customized interpolation: subst

```
subst ?-nobackslashes? ?-nocommands? ?-novariables? STRING
```

The `subst` command offers a more flexible form of string interpolation with control of the types of substitution that will take place. The command performs backslash, variable and command substitution on the *`STRING`* argument in the same manner as the Tcl parser and returns the result.

When the `subst` command is invoked two rounds of substitution take place, first by the Tcl parser, and then by the `subst` command itself.

```
subst "(\\t)" → (       ) ❶
subst {(\\t)} → (\t) ❷
subst {(\t)}  → (       ) ❸
```

❶ `subst` sees `(\t)` as Tcl parser does one round of substitution
❷ `subst` sees `(\\t)` as the braces prevent substitution by the Tcl parser
❸ `subst` sees `(\t)`

The examples below use `{}` to prevent the Tcl parser from making substitutions so as to make the `subst` command behaviour clear.

```
set var 2                                  → 2
subst {$var+$var\t=\t[expr {$var+$var}]} → 2+2     =       4
```

The `-nobackslashes`, `-nocommands` and `-novariables` options provide additional control. These options selectively prevent substitution of backslash sequences, commands and variables respectively.

```
subst {$var+$var\t= [expr {2+2}]}                → 2+2     = 4
subst -nobackslashes {$var+$var\t= [expr {2+2}]} → 2+2\t= 4
subst -nocommands {$var+$var\t= [expr {2+2}]}    → 2+2     = [expr {2+2}]
subst -novariables {$var+$var\t= [expr {2+2}]}   → $var+$var       = 4
```

There are some subtleties in the interaction among the various options to `subst` or when commands raise an error. Consider

```
% subst -novariables {The sum $var+$var\t=\t[expr {$var+$var}]}
→ The sum $var+$var       =       4
```

and notice how the variables inside the `expr` expression have been substituted despite the `-novariables` option. See the Tcl reference documentation for such special cases.

The flexibility of `subst` is the basis for several text transformation libraries; for example see substify[4] in the Tcler's Wiki which generates HTML from a template consisting of Tcl and HTML fragments in just a few lines.

4.20. Formatting strings: format

```
format FORMATSTRING ?arg1 arg2…?
```

The `format` command is intended for situations where you need to construct strings with precise representation and structure. For example, you may need to write floating point values to a CSV file to be imported by another application which requires exactly two decimal places or a report where values have to fit a specific column width.

FORMATSTRING is a template containing literal text as well as *field specifiers* that are placeholders for the values supplied as arguments. The command returns *`FORMATSTRING`* with field specifiers replaced by formatted argument values.

```
% format "%d times %#x is %e" 10 10 100
→ 10 times 0xa is 1.000000e+02
```

Here `%d`, `%#x` and `%e` are field specifiers that control how the numbers are formatted.

A field specifier is delimited by a literal `%` character and a conversion character that specifies the type. Within those delimiters, it may contain in order

- an optional XPG3 specifier to change argument order
- an optional sequence of flag characters controlling justification, padding and alternative representations

[4] https://wiki.tcl-lang.org/page/Templates+and+subst

- an optional minimal width
- an optional precision or bound
- an optional size modifier for the value

All the parts above are optional except the starting `%` and ending conversion character.

4.20.1. Conversion characters

The mandatory conversion character controls the type of conversion to be applied to the corresponding argument. The conversions may be classified as string, integer and floating point depending on the type of value to be formatted.

The format specifiers for integer arguments are shown in Table 4.1. In the absence of a size modifier (Section 4.20.6), 32-bit values are assumed.

Table 4.1. Integer specifiers for format

Type	Formats as ...	Example
`b`	Binary integer.	`format %b 42 → 101010`

`d`, `i`	Signed decimal integer.	`format %d 0xffffffff → -1`
		`format %d 42 → 42`
		`format %i 0xffffffff → -1`
		`format %i 42 → 42`

`o`	Octal integer.	`format %o 42 → 52`
`p`	Same as `0x%zx`. Not available in Tcl 8.6.	
`u`	Unsigned decimal integer.	`format %u 0xffffffff → 4294967295`
`x`, `X`	Lower/upper case hexadecimal integer.	`format "%x,%X" 42 42 → 2a,2A`

Table 4.2 shows the specifiers for string and character conversion.

Table 4.2. String specifiers for format

Type	Formats as ...	Example
`s`	String. Useful with modifiers like field widths (Section 4.20.3).	`format %s 0xffffffff → 0xffffffff`
`c`	Character. Argument is an integer Unicode code point.	`format %c 42 → *` `format %c 0x662D → 昭`
`%`	A literal percent character. Does not consume an argument.	`% format "Tcl! 100%% sugar-free!"` `→ Tcl! 100% sugar-free!`

Table 4.3 shows the specifiers for formatting floating point numbers.

Table 4.3. Floating point specifiers for format

Type	Formats as ...	Example
`a`, `A`	Lower/upper case hexadecimal form *0x1.yyy***p**±*zz* where *yyy* has the width given by the precision (default 13). Not in Tcl 8.6.	`format %a 42 → 0x1.5000000000000p+5` `format %A 42 → 0x1.5000000000000p+5`
`f`	Signed decimal *xx.yyy*, width of *yyy* based on precision (default 6).	`format %f 4.2e1 → 42.000000`
`e`, `E`	Scientific representation *x.yyy***e**+-*zz* or *x.yyy***E**+-*zz*, width of *yyy* based on precision (default 6). If precision is 0, no decimal point is output.	`format %e 42 → 4.200000e+01` `format %E 42 → 4.200000E+01`

`g`, `G`	As `%e` or `%E` respectively if the exponent is less than -4 or at least the specified precision; otherwise, as `%f`. Trailing 0's and decimal point are omitted.	`format %g 420e-1  → 42`
		`format %G 420e-1  → 42`
		`format %G 420e-10 → 4.2E-08`

4.20.2. XPG3 format position specifiers

By default, format specifiers and supplied arguments are matched in the order they appear. For example, in the command below `%d` and `%s` get matched up with `31` and `January`.

```
% format "There are %d days in %s." 31 January
→ There are 31 days in January.
```

This is not always desirable behavior. One example is the formatting of strings for different languages. Localization typically involves passing an identifier to the message catalog (Section 9.3.2) which returns the appropriate string for the language. The insertion point for each argument value depends on the language's grammar. For example, consider this simple procedure for printing a localized message for the days in a month.

```
set english "There are %d days in %s." ❶
set canadian "%s has %d days, eh!"
proc print_days {fmt month days} {
    puts [format $fmt $days $month]
}
print_days $english January 31
→ There are 31 days in January.
```

❶ Assume returned from localized message catalogs

This works for English. However, we have problems in Canada because the order of arguments no longer matches the order of specifiers.

```
% print_days $canadian January 31
Ø expected integer but got "January"
```

The XPG3 position specifiers address this issue. A position specifier follows the leading `%` and consists of the numeric position of the corresponding argument in the argument list followed by a `$` character.

The message catalog strings in the above example would be written as

```
set english {There are %1$d days in %2$s.}
set canadian {%2$s has %1$d days, eh!}
```

Now the order of arguments that is passed to `format` is fixed while still allowing for the insertions to take place in a different order. The Canadians are now happy.

```
print_days $english January 31  → There are 31 days in January.
print_days $canadian January 31 → January has 31 days, eh!
```

Argument indices may be repeated if necessary. For example,

```
% format {%1$d == 0x%1$x == 0o%1$o} 42
→ 42 == 0x2a == 0o52
```

repeats a single integer argument thrice with different formats.

XPG3 position specifiers must be present in **all** specifiers in a format string or **none**.

4.20.3. Specifying minimum field widths

The minimal width part of a specifier mandates a minimal number of characters in the inserted value, useful when formatting tabular data to match desired column widths. The width can be specified as either a number or the `*` character when supplied as an additional argument.

```
% format "(%d)" 10
→ (10)
% format "(%8d)" 10
→ (      10)
% format "(%*d)" 8 10
→ (      10)
```

4.20.4. Format flags

The flags component controls representation as shown in Table 4.4.

Table 4.4. Flag component in format specifiers

Flag	Description	Example
`-`	Forces padding on the right when a minimum width is in effect.	`format (%8d) 10  → (      10)` `format (%-8d) 10 → (10      )`
`0`	Pads with 0's instead of spaces.	`format (%08d) 10 → (00000010)`
`+`	Prefixes positive numbers with `+`.	`format (%+d) 10 → (+10)`
Single space	Includes a single space before a number **unless a sign is present**.	`format "(% d)" 10  → ( 10)` `format "(% d)" -10 → (-10)`

`#`	For binary, octal and hexadecimal fields, the flag adds an appropriate prefix, for example `0x` for hexadecimal. For floating point conversions, it forces inclusion of a decimal point and in the case of `g` and `G`, inclusion of trailing zeroes.	`format %#x 10 → 0xa`
		`format %#X 10 → 0xA`
		`format %#o 10 → 0o12`
		`format %#b 10 → 0b1010`
		`format %g 10  → 10`
		`format %#g 10 → 10.0000`

4.20.5. Precision specifier

The fourth part, also optional, of a conversion specifier consists of a period (`.`) followed by a numeric precision or a `*` in which case the precision is supplied through an additional argument.

For string conversion, the precision is the **maximum** number of characters to produce; for integer conversion, it is the **minimum**.

```
format %.2s abc  → ab ❶
format %.5d 10   → 00010 ❷
format %.*d 4 10 → 0010 ❸
```

❶ At **most** two characters
❷ At **least** five characters
❸ At **least** four characters as specified by additional argument

For the `e`, `E` and `F` conversions, precision controls the number of digits to the right of the decimal point.

```
format %f 1.12999   → 1.129990 ❶
format %.2f 1.12999 → 1.13
```

❶ Note by default 6 digits printed

For `g` and `G` conversions, it specifies the *total* number of digits output except trailing 0's are still omitted if the `#` flag is not specified.

```
format %g 1.12999   → 1.12999
format %.2g 1.12999 → 1.1
format %.2g 1.01    → 1
format %#.2g 1.01   → 1.0
```

4.20.6. The size modifier

The size modifier, shown in Table 4.5, controls truncation of integer arguments.

Table 4.5. Size modifiers for format

Type	Description
Not present.	Values are truncated to 32 bits.
`h`	Truncates to a 16-bit value.
`I`, `I32`	Truncates to a 32-bit value. Not available in Tcl 8.6.
`I64`	Truncates to a 64-bit value. Not available in Tcl 8.6.
`l`, `j`, `q`	Truncates to the same range as the `tcl::mathfunc::wide` function, which is at least 64-bits. `j` and `q` are not available in Tcl 8.6.
`ll`, `L`	Do not truncate. `L` is not available in Tcl 8.6.
`t`, `z`	Truncates to the range indicated by the `pointerSize` element of the `tcl_platform` array. Not available in Tcl 8.6.

Examples are given below. These use the `x` type specifier for clarity but size modifiers apply regardless. Note the result will depend on the platform and architecture in some cases. The shown output is for 64-bit Windows.

```
set i32 0x7fffffff          → 0x7fffffff
set i64 0x7fffffffffffffff  → 0x7fffffffffffffff
set i65 0x10000000000000001 → 0x10000000000000001
format %hx $i32             → ffff
format %x $i64              → ffffffff
format %x $i65              → 1
format %lx $i64             → 7fffffffffffffff
format %lx $i65             → 1
format %llx $i64            → 7fffffffffffffff
format %llx $i65            → 10000000000000001
```

4.21. Parsing strings: scan

```
scan INPUTSTRING FORMATSTRING
scan INPUTSTRING FORMATSTRING VAR1 ?VAR2 …?
```

There are two Tcl commands that are commonly used in parsing. One of them, `regexp`, is based on regular expressions and we describe it in Section 10.1. The other one, `scan`, is similar to the `sscanf` library function in C and described here.

The `format` command (Section 4.20) we saw earlier generates a string composed from input values formatted as per a specification. Conversely, the `scan` command provides a means to parse strings that are known to be in a specific format, converting its substrings to values of a specific type.

The command takes one of two forms, one where the parsed values are returned as the result of the command and the other where they are stored in variables.

In both forms, *INPUTSTRING* is the string to be parsed while *FORMATSTRING* controls the parsing. The command works by iterating over each character in *FORMATSTRING* and matching it against *INPUTSTRING* as follows:

- If the format character is a space or a tab, the command skips over zero or more consecutive whitespace characters in the input string.
- If the format character is `%`, it is the start of a conversion specifier. The input string is parsed based on the specifier and the value extracted as per specifier type. This is detailed below.
- Any other format character must exactly match the character in the input string in which case the scan continues with the next character. Otherwise the scan is ended and any remaining characters in *INPUTSTRING* are ignored. Note that this is not treated as an error.

If the first form of the command is used where only two arguments are present, the extracted values are returned as the result of the command. We will refer to this as the inline form. In the second form, the additional arguments are treated as names of variables into which the extracted values are to be stored. In this case, the return value from the command is the number of conversions performed. For example,

```
set input "pi    =  3.14159"         → pi    =  3.14159
scan $input "%s = %f"                → pi 3.14159
scan $input "%s = %f" name value     → 2
puts "The value of $name is $value." → The value of pi is 3.14159.
```

The parsing in the above example proceeds as follows:

- the `%s` format string is matched with `pi` in the input string
- the space character is matched against multiple spaces in the input string
- the `=` literal in the format string exactly matches the one in the input string
- spaces are skipped again
- finally the `%f` format results in the parsing of `3.14159` as a floating point value

The format string is composed of literal characters and conversion specifiers. A conversion specifier is a string of characters composed of the following parts or components **in order**.

- The character `%`
- An optional XPG3 specifier
- An optional maximum substring width

- An optional size modifier
- The conversion character

Note that only the starting `%` and the conversion character are mandated.

4.21.1. Conversion characters

The conversion character controls the type of conversion to be performed.

The integer conversion characters are shown in Table 4.6.

Table 4.6. Integer specifiers for scan

Character	Description	Example
`d`, `u`	Signed and unsigned decimal integer.	`scan -100 %d → -100` `scan 0100 %u → 100`
`x`, `X`	Hexadecimal integer	`scan 0100 %x → 256`
`o`	Octal integer	`scan 0100 %o → 64`
`b`	Binary integer	`scan 0100 %b → 4`

The string conversion characters are shown in Table 4.7.

Table 4.7. String specifiers for scan

Character	Description	Example
`s`	Parse as a string up to the next **white space** character.	`scan "foo bar" %s → foo`
`c`	Convert a character to its Unicode code point value	`scan A %c → 65`
[*CHARS*], [^*CHARS*]	Matches sequence of characters in or not in *`CHARS`*.	`scan "cab123" {%[abc]}  → cab` `scan "cab123" {%[^123]} → cab`
`%`	Matches the percent character itself. Not included in results.	`scan "10% off!" "%d%% off" → 10`

For floating point conversion, any of `f`, `e`, `E`, `g` and `G` can be used. These all have the same effect and are interchangeable.

```
scan 100 %f       → 100.0
scan 12.34e-56 %g → 1.234e-55
```

The final conversion specifier is `n` which is a special case in that it does not parse the input string at all. Instead it returns the number of characters of the input string that have been parsed so far.

```
scan "100      200" "%d %n%d" → 100 8 200
```


The `n` conversion is useful to determine the start of the next scan when scanning incrementally through the input string.

A conversion specifier keeps matching each successive character in the input string as long as the character is valid for the conversion. For example, compare the following conversions:

```
scan 123ABX %d%s → 123 ABX
scan 123ABX %x%s → 74667 X
```

The difference arises from `A` and `B` being valid hexadecimals but invalid decimals.

4.21.2. Scan termination

There are several conditions under which the `scan` command will terminate further processing of the input string.

- The end of the input string is reached.
- A conversion fails.
- All conversions succeed.

The command results are different in each case as we illustrate below. Our examples use the `%d` specifier which attempts conversion of the input substring to an integer value.

In the first scenario, the end of the input string is reached before any conversions are **attempted** (although the reference manual says **performed**). In this case, `scan` returns an empty list in the inline version of the command. If variables are specified for storing the result, the command returns `-1` and no variables are assigned.

```
scan abc abc%d     → (empty)
scan abc abc%d val → -1
info exists val    → 0
```

In the above example, the `abc` in the input string matches the `abc` in the format string. At that point, no further processing is done because no input remains.

In the second scenario, shown below, the processing stops before the end of the input string is reached because a scan conversion fails. In this case, the inline version returns a list of the same length as the number of scan specifiers in the format string. The elements in the returned list corresponding to conversions that failed, or were not attempted due to scan termination, are set to the empty string.

```
scan abcX %d                       → {}
scan "abc10 def 20" "abc%d %d %d" → 10 {} {}
```

Compare the first result in this scenario with that in the first scenario above. There the command returned an empty list. Here it returns a list containing one element which is the empty string[5] corresponding to the single format specifier present. Similarly, in the second result, the last two elements in the returned list are empty as the second conversion failed thereby terminating the parse.

If variables are specified in this scenario, the return value of the command is the number of conversions performed. The variables corresponding to failed conversions will not be modified or created.

```
% scan abcX %d vara ❶
→ 0
% info exists vara
→ 0
% scan "abc10 def 20" "abc%d %d %d" vara varb varc ❷
→ 1
% set vara
→ 10
% info exists varb
→ 0
```

❶ Note the difference from the first scenario (input termination before any conversion attempts) above where the return value was `-1`
❷ Only one conversion successful

Because the parsing was terminated by the failed match on the second `%d` specifier, variables `varb` and `varc` are not assigned.

The final scenario is when all conversions succeed. The scan is then terminated irrespective of whether there are any remaining characters in either the input or the format string. In the inline version, the returned value is a list each element of which is the value resulting from a successful conversion for the corresponding field specifier. In the non-inline version, the return value equals the number of field specifiers and each variable contains the corresponding value.

```
scan "abc10 15 20xyz" "abc%d %d %d"             → 10 15 20
scan "abc10 15 20" "abc%d %d %d" vara varb varc → 3
set varc                                        → 20
```

When variables are specified, their number must match the number of successful conversions else the command will raise an error exception.

4.21.3. XPG3 scan position specifier

By default extracted values are returned or stored in the passed variables in the order they are seen in the input string. The XPG3 position specifier allows this to be changed. This is analogous to the `format` command's XPG3 position specifier (Section 4.20.2).

[5] Tcl represents empty elements within a list as empty braces.

The position specifier immediately follows the `%` at the start of a conversion specifier. It consists of either a number followed by a `$` character or a single `*` character. In the former case, the number indicates the position the extracted value should occupy in the result.

```
% scan "first second" "%s %s"
→ first second
% scan "first second" {%2$s %1$s} ❶
→ second first
% scan "first second" {%2$s %1$s} varA varB
→ 2
% puts "varA=$varA, varB=$varB"
→ varA=second, varB=first
```

❶ Note use of `{}` to protect the `$` from interpretation by the Tcl command parser

A `*` character in a XPG3 position specifier indicates that the input string should be parsed as per the conversion specifier but the extracted value should not be returned or stored into the output variables.

```
% scan "100 200 300" {%d %*d %d}
→ 100 300
```

The XPG3 component must be present in **all** specifiers or **none**.

Position specifiers can be used in `scan` to help with localization similar to their use with `format` (Section 4.20.2). However this is less common as parsing of localized strings is generally a much more complex process than their generation.

4.21.4. Specifying maximum widths

The number of characters consumed by a conversion can be limited by including the numeric width in the specifier.

```
% scan 12345 "%d%s"
→ 12345 {}
% scan 12345 "%2d%s" ❶
→ 12 345
```

❶ The `d` conversion should consume at most 2 input characters

Some file formats are based on fixed lengths for each field in a line representing a data record. This width modifer is useful in such cases.

4.21.5. The size modifier

The size modifier defines the permitted range of an integer argument. On overflow, the maximum value possible for that size is stored. Possible values are shown in Table 4.8 below.

Table 4.8. Integer size modifiers for scan

Character	Description
Not present	Defaults to `h`.
`h`	An `int` value, generally 32 bits on most platforms. Overflows store 0x7fffffff.
`l`, `j`, `q`	The same range as the `tcl::mathfunc::wide` function, which is at least 64-bits with overflows storing 0x7fffffffffffffff. `j` and `q` are not available in Tcl 8.6.
`ll`, `L`	Arbitrary precision with no overflow. In Tcl 8.6, `L` was equivalent to `l`.
`t`, `z`	Range indicated by the `pointerSize` element of the `tcl_platform` array. Not available in Tcl 8.6.

The examples below illustrate the difference between the various size modifiers.

```
% set val 77777777777777777777777777777777777777
→ 77777777777777777777777777777777777777
% scan $val %d
→ 2147483647
% scan $val %hd
→ 2147483647
% scan $val %ld
→ 9223372036854775807
% scan $val %Ld
→ 77777777777777777777777777777777777777
% scan $val %lld
→ 77777777777777777777777777777777777777
```

4.22. Comparing strings: string equal|compare

```
string equal ?-nocase? ?-length COUNT? STRING1 STRING2
string compare ?-nocase? ?-length COUNT? STRING1 STRING2
```

The `string equal` command returns `1` if *`STRING1`* and *`STRING2`* are identical and `0` otherwise. The comparison is case-sensitive unless the `-nocase` option is specified.

```
set s Hello                    → Hello
string equal $s Hello          → 1
string equal hello $s          → 0
string equal -nocase hello $s → 1
```

The `-length` option limits the comparison to the first *`COUNT`* characters.

```
string equal "Hello World!" "Hello Universe!"           → 0
string equal -length 5 "Hello World!" "Hello Universe!" → 1
```

The `string compare` command compares the passed strings for lexicographical ordering. The command returns `-1`, `0` or `1` depending on whether the first argument is lexicographically less than, equal to, or greater than the second.

```
string compare abcd BCDE         → 1
string compare -nocase abcd BCDE → -1
string compare 2 10              → 1 ❶
```

❶ Compared as strings, **not** numbers

As for `string equal`, the `-nocase` and `-length` options select a case-insensitive comparison and impose a limit on the number of characters compared.

4.23. String validation: string is

```
string is CLASS ?-strict? ?-failindex VAR? STRING
```

The `string is` command validates that a string belongs to a given class of values. The command returns `1` if *`STRING`* belongs to the class specified by *`CLASS`* and `0` otherwise.

In the absence of the `-strict` option, the empty string `""` is treated as a valid value for any class.

```
string is integer ""          → 1 ❶
string is integer -strict ""  → 0 ❷
```

❶ Empty strings are accepted by default…
❷ …unless the `-strict` option is specified

The `-failindex` option is used to retrieve the index in the string of the first character that does not belong to the specified class. If the command returns `0`, the variable *`VAR`* is set to this index or -1 if the failure location could not be identified. The variable is not modified if the command returns `1`.

```
% string is xdigit -failindex charpos abcqdef  ❶
→ 0
% set charpos
→ 3
```

❶ `charpos` contains the failing character index

Tcl's regular expression facilities (Chapter 10) provide an alternate means of validating strings using character classes.

The possible values of *`CLASS`* are shown in Table 4.9.

Table 4.9. String validation classes

Class	Description
`alnum`	Alphanumeric Unicode characters.
`alpha`	Alphabetic Unicode characters.
`ascii`	Ascii characters.
`boolean`	Any string interpretable as a boolean value. Accepted values are the union of the ones listed for `false` and `true`. Note the command returns `0` for integers other than `0` and `1` though they are treated as valid booleans in numeric expressions.
`control`	Unicode control characters.
`dict`	Any representation of a Tcl dictionary (Section 6.1).
`digit`	Unicode digits.
`double`	Any representation of doubles.
`entier`	Any representation of integers of arbitrary size.
`false`	Any string interpretable as a boolean `false` value. This includes `0`, `false`, `no`, `off` and their upper case or abbreviated forms.
`graph`	Unicode printing characters excluding whitespace.
`integer`	Same as `entier`. Tcl 8.6 and earlier: only accepts 32-bit integers.
`list`	Any string that can be interpreted as a valid Tcl list.
`lower`	Lower case Unicode characters.
`print`	Unicode printing characters including whitespace.
`punct`	Unicode punctuation characters.
`space`	Unicode whitespace characters.
`true`	Any string interpretable as a boolean `true` value. This includes `1`, `true`, `yes`, `on` and their upper case or abbreviated forms.
`upper`	Upper case Unicode characters.
`wideinteger`	Any Tcl representation of 64-bit integer values.
`wordchar`	Alphanumeric characters and connector punctuation such as underscore.
`xdigit`	Lower or upper case hexadecimal characters.

The following examples illustrate use of the command:

```
string is integer -10                → 1
string is wideinteger 0x777777777777 → 1
string is double 2.1828              → 1
string is boolean 2                  → 0 ❶
```

❶ Integers other than `0`/`1` are **not** treated as boolean though they are accepted as booleans in numeric expressions.

The `list`, `dict` and numeric classes accept surrounding whitespace in the string.

```
string is double "  -5e+10   "    → 1
string is list " a\ b c {d e}   " → 1
```

The commands for numeric classes also check for overflow and underflow, returning 0 in these cases and setting the `-failindex` variable, if specified, to -1.

```
% string is entier -failindex failpos 99999999999999999999999999999999 ❶
→ 1
% string is wideinteger -failindex failpos 99999999999999999999999999999999
→ 0
% puts $failpos
→ -1
```

❶ Entiers have infinite precision

The alphanumeric character classes such as `alnum`, `digit` etc. are not limited to ASCII. So for example the Unicode character U+096D (Devanagari 7) is a valid digit.

```
string is digit \u096d → 1
```

Note however that

```
string is integer \u096d → 0
```

because even though it is a Unicode digit, non-ASCII characters cannot be used to represent an integer in Tcl.

4.24. Glob pattern matching: string match

```
string match ?-nocase? PATTERN STRING
```

The `string match` command matches a string against a glob pattern, returning `1` on a match and `0` otherwise. In addition to `string match`, glob patterns are also used by many other commands such as `switch` (Section 3.7), `lsearch` (Section 5.22), `glob` (Section 12.3.5) and others.

Glob patterns described here should not be confused with regular expressions described in Chapter 10.

Each character in a glob pattern must match the corresponding character in the string being matched except for the *wildcard* characters summarized in Table 4.10.

Table 4.10. Pattern matching characters

Character	Description
`*`	Matches any number (including zero) of arbitrary characters.
`?`	Matches exactly one arbitrary character.
`[…]`	Matches any of the characters included within the brackets. A range of characters, e.g. `a-z`, can also be specified.
`\`	Disables special treatment of the following character such as `*` or `?` allowing inclusion of glob-sensitive characters in glob patterns.

The `*` pattern matches an arbitrary number of characters.

```
string match f*r fun      → 0
string match f*r fur      → 1
string match f*r* fury    → 1
string match f*r* furious → 1
```

The `?` character on the other hand matches **exactly** one character.

```
string match f?r? fur     → 0
string match f?r? fury    → 1
string match f?r? furious → 0
```

Character sets can be used to match exactly one character from a set.

```
string match {[a-f]*} boo    → 1
string match {[a-f]*} zoo    → 0
string match {[a-zA-Z]*} Zoo → 1
string match {[az]*} zoo     → 1
```

Backslash escaping treats special characters as literals.

```
string match a*d abcd    → 1
string match {a\*d} abcd → 0
string match {a\*d} a*d  → 1
```

Notice from the examples above that we use braces to protect the pattern in cases where it contains characters like `\` or `[]` that are special to the Tcl parser.

Use of the `-nocase` option triggers case insensitive matching.

```
string match {[a-z]*} Boo         → 0
string match -nocase {[a-z]*} Boo → 1
```

4.25. Matching shared prefixes: ::tcl::prefix

```
::tcl::prefix all LIST PREFIX
::tcl::prefix longest LIST PREFIX
::tcl::prefix match ?-exact? ?-message MESSAGE? ?-error OPTIONS? LIST PREFIX
```

The `::tcl::prefix` command ensemble facilitates checking if a string is a prefix of one or more strings in a set. This operation is commonly used in implementation of commands that accept abbreviations that are unique prefixes of options and subcommands. The command lies in the `tcl` namespace, not the global level.

The `tcl::prefix all` command returns a list of all strings from *`LIST`* that begin with *`PREFIX`* or an empty list if no such strings are found. Interactive shells use this to display valid choices for completion when the user types in the first few letters of a command or option. For example, if *`LIST`* is a subset of the string classes used for the `string is` command,

```
::tcl::prefix all {alnum alpha digit integer} al → alnum alpha
```

The `tcl::prefix longest` command returns the longest possible prefix common to all strings in *`LIST`*. In scenarios like command completion, this provides an easy means to fill in as many characters as possible from the valid choices in *`LIST`* that begin with *`PREFIX`*.

```
% ::tcl::prefix longest {radix range repeat replace} re
→ rep
```

The most commonly used member of the ensemble is `tcl::prefix match`. If *`PREFIX`* is a prefix of exactly one string in *`LIST`*, that string is returned as the result of the command. Moreover, if *`PREFIX`* matches a string **in its entirety**, it is returned even if it also happens to be a prefix of another.

```
% ::tcl::prefix match {radix range repeat replace} rad
→ radix
% ::tcl::prefix match {-ignore -ignorewarnings -ignoreerrors} -ignore
→ -ignore
```

By default, an error is generated if the number of matches is not exactly one.

```
% ::tcl::prefix match {radix range repeat replace} ra
Ø ambiguous option "ra": must be radix, range, repeat, or replace
% ::tcl::prefix match {radix range repeat replace} rap
Ø bad option "rap": must be radix, range, repeat, or replace
```

The `-message` option changes how the error message refers to the word being checked.

For example, note below how the error message now says `command` instead of `option`.

```
% ::tcl::prefix match -message command {radix range repeat replace} ra
Ø ambiguous command "ra": must be radix, range, repeat, or replace
```

The `-error` option changes the behavior with respect to failed matches. If specified as an empty string, the command will return an empty string on the above failures instead of raising an exception.

```
::tcl::prefix match -error "" {radix range repeat replace} ra → (empty)
```

If not an empty string, the value passed for the `-error` option must be in the form of a return options dictionary (Section 15.2.3) that can be used to change error information.

```
% ::tcl::prefix match {radix range repeat replace} rap
Ø bad option "rap": must be radix, range, repeat, or replace
% set errorCode ❶
→ TCL LOOKUP INDEX option rap
% ::tcl::prefix match -error {-errorcode {BADOPT RTFM}} {radix range repeat \
    replace} rap
Ø bad option "rap": must be radix, range, repeat, or replace
% set errorCode
→ BADOPT RTFM
```

❶ Global error code is set to a default error code.

The `-exact` option specifies that the *`PREFIX`* must be an exact match, not just a prefix. Its effect can be seen through the following commands.

```
% ::tcl::prefix match {radix range repeat replace} ran
→ range
% ::tcl::prefix match -exact {radix range repeat replace} ran
Ø bad option "ran": must be radix, range, repeat, or replace
```

Here is a simple example of how `tcl::prefix` is commonly used for implementing subcommands and options in procedures.

```
proc transform {s cmd} {
    switch -exact -- [tcl::prefix match {lower upper reverse} $cmd] {
        lower { string tolower $s }
        upper { string toupper $s }
        reverse { string reverse $s }
    }
}
```

Then we can call the command using abbreviations.

```
% transform foo rev
→ oof
% transform foo bogus ❶
Ø bad option "bogus": must be lower, upper, or reverse
```

❶ Error messages for free!

5

Lists

> The human animal differs from lesser primates in his passion for lists.
>
> — H. Allen Smith

A *list* in Tcl is an ordered collection, or sequence, of values and serves a purpose similar to that of arrays indexed by integers in other languages. Lists can be nested, allowing their use for construction of more elaborate data structures like trees.

5.1. Basic list construction: list

```
list ?VALUE VALUE …?
```

The `list` command is the basic mechanism for constructing a list. It accepts an arbitrary number of arguments and returns a list containing those values.

For example,

```
% set items [list "Item One" Item_with_no_spaces "Item Three"]
→ {Item One} Item_with_no_spaces {Item Three}
% set numbers [list [set i 0] [incr i] [incr i]]
→ 0 1 2
```

5.2. List literals

List values can also be constructed as literals via their string representation. For example, the following assigns the string `One Two Three` to the variable `mylist`.

```
set mylist {One Two Three} → One Two Three
```

Any command that operates on lists will interpret this string as a list with three elements `One`, `Two` and `Three`.

```
llength $mylist    → 3
lindex $mylist end → Three
```

It is common practice to define literal lists using braces as opposed to quotes. **However, the use of braces does not in any way or form imply that the value is a list.** As described in Section 3.3, both braces and quotes denote string literals with the difference being the

treatment of special characters. It is the commands expecting an argument to be a list that parse the passed string, converting it to a list value.

Tcl maintains an internal representation of the list just as it does for numeric values. As long as the values are manipulated using list commands, there is never a performance impact of conversions to and from string representations.

The following two assignments result in the same list.

```
% set la {{Item One} Item_with_no_spaces "Item Three"}
→ {Item One} Item_with_no_spaces "Item Three"
% set lb "\"Item One\" Item_with_no_spaces {Item Three}"
→ "Item One" Item_with_no_spaces {Item Three}
% foreach a $la b $lb { puts "$a == $b" }
→ Item One == Item One
  Item_with_no_spaces == Item_with_no_spaces
  Item Three == Item Three
```

Multiple string representations

One point to be noted about lists is that different strings may represent the same list. Just as `0xa` and `10` are different strings representing the same number in hexadecimal and decimal format, strings that do not compare equal may represent the same list. The two strings below are valid representations of the same list though they do not compare equal as strings.

```
set la "     this     is     a     list      "
set lb "this is a list"
foreach a $la b $lb { puts "$a == $b" }
→ this == this
  is == is
  a == a
  list == list
```

Although a list may have many string representations, Tcl always ensures that the **individual** elements of lists are retrieved in exactly the format that they were created. In other words, adding an element to a list does not change the element's string representation and the following invariant holds for all list operations:

```
[lindex [linsert LIST INDEX ELEM] INDEX] eq ELEM
```

where `eq` is the string equality operator. Essentially what that means is that inserting and then retrieving an element will not change its string representation.

Parsing string representations of lists

Commands which operate on lists convert the string forms into lists by parsing them in the same manner as described in Section 3.1 for parsing commands into words except that no variable and command substitution is performed. In particular, the list commands will still perform **backslash substitution** when converting a string value to a list.

Because the parsing and interpretation of a string as a list is often a source of confusion, let us walk through another example. Consider the following two commands to retrieve the first element of a list using `lindex`.

```
lindex "a\ b c" 0 → a
lindex {a\ b c} 0 → a b
```

Note that the characters within the quotes and braces are identical but the results are different. If you understand why, skip to the next section. Else read on.

Following the rules described in Chapter 3, the Tcl parser treats backslashes inside the quoted string as escapes for the next character. So in the first case, it replaces the backslash and following space with a single space character. Consequently, the `lindex` command receives the string value

```
a b c
```

as its first argument. As we said earlier, the list based commands apply the same rules as the Tcl parser for breaking up a string into words. Thus when `lindex` parses the argument following those rules, the string gets parsed into three words (list elements), `a`, `b` and `c`. These form the elements of the constructed list whose first element is returned.

On the other hand, since backslash replacement is not done inside braces, in the second case `lindex` receives the string value

```
a\ b c
```

Now when `lindex` parses the value, it uses the same backslash substitution rules whereby the backslash protects the following space from being treated as a word separator. Consequently the string is parsed as just two words — `a b`, and `c` — which then make up the constructed list.

In essence, when enclosed in double quotes, the argument undergoes one round of backslash substitution by the Tcl parser and then a **second** round when the variable's value is parsed by the `lindex` command to convert it to a list. When the string is enclosed in braces on the other hand, it is protected from substitution by the Tcl parser and only undergoes the backslash substitution by `lindex`.

In most cases, the limited substitution inside braces makes their use more convenient than the use of quotes when defining lists. Even then, use of braced strings as list literals should only be used in simpler cases. For more complex lists, it is strongly recommended that the `list` command be used.

Do **not** use string interpolation in lieu of the `list` command to construct lists or use string commands to manipulate list values. Although string interpolation and commands may seem to work in some cases, they will not give the correct result when the values contain whitespace or other special characters.

5.3. List indices

Elements within a list are referenced by their position in the list using 0-based *list indices* as described for strings. Refer to Section 4.2 for details including index arithmetic. **Unless mentioned otherwise, list commands interpret `end` as the index of the last element**.

5.3.1. Nested list indices

Elements in a list may themselves be interpreted as lists. To facilitate operations on such nested lists, many list commands accept a sequence of indices in place of a single index. The indices act like a "path" through the nested list where the elements of the index list identify an element at each nesting level. We will see examples as we proceed through this chapter.

5.4. Retrieving elements by position: lindex

```
lindex LIST ?INDEX …?
```

The `lindex` command returns the element at a specific position in the list.

In the simple case where only one index is specified, the command returns the element at that position in the list.

```
set l [list a [list b0 b1 b2] c] → a {b0 b1 b2} c
lindex $l 0                      → a
lindex $l 1                      → b0 b1 b2
```

The number of indices determines the nesting depth of `LIST`. For each provided index, the command will burrow an additional level into the element at that index effectively treating the indices as a path through the nested list.

```
lindex $l       → a {b0 b1 b2} c ❶
lindex $l 1     → b0 b1 b2
lindex $l 1 end → b2
```

❶ Nesting depth of 0 (no indices specified), entire list argument is returned.

The multiple indices may be provided as a single list or as independent arguments. Thus the following are equivalent.

```
lindex $l 1 end   → b2
lindex $l {1 end} → b2
```

Note that `lindex` will return an empty string, not raise an error, if passed an index that is negative or greater than the list size.

5.5. Extracting elements by position: lpop

```
lpop VARNAME ?INDEX …?
```

The `lpop` command extracts and returns the element at a specific, possibly nested, position in a list. Unlike `lindex`, `lpop` is passed the name of a variable containing the list operand and not the list value. It operates on the list stored in the variable, removing the indexed element from the list and storing the result back in the variable.

```
set lst {a b c d e f g}                 → a b c d e f g
puts [lpop lst],$lst                    → g,a b c d e f ❶
puts [lpop lst end-2],$lst              → d,a b c e f ❷
puts [lpop lst [lsearch $lst b]+1],$lst → c,a b e f ❸
set nested {a b {c d {e f}} g}          → a b {c d {e f}} g
puts [lpop nested end-1 end 0],$nested  → e,a b {c d f} g ❹
```

❶ *INDEX* defaults to `end`
❷ Normal index arithmetic
❸ Remove element after first `b`
❹ Remove first element within last element of penultimate outer list element

Unlike `lindex`, `lpop` will raise an error if an index is out of range.

5.6. Retrieving a sublist: lrange

```
lrange LIST FIRST LAST
```

The `lrange` command returns a list containing all elements between indices *`FIRST`* and *`LAST`* (both inclusive) in *`LIST`*.

```
% set monthlyDownloads {120 110 130 100 90 85 92 105 114 140 156 190}
→ 120 110 130 100 90 85 92 105 114 140 156 190
% lrange $monthlyDownloads 0 2 ❶
→ 120 110 130
% tcl::mathop::+ {*}[lrange $monthlyDownloads end-2 end] ❷
→ 486
```

❶ Downloads by month in the first quarter
❷ Total downloads in the last quarter

5.7. Retrieving leading elements: lassign

```
lassign LIST ?VARNAME VARNAME …?
```

The `lassign` command sequentially assigns the leading elements in a list to the specified variables returning a new list (possibly empty) containing the rest. If there are fewer elements in the list than variables specified, the additional variables are set to the empty string.

```
set remaining [lassign {A B C D} x y]   → C D
puts "x=$x, y=$y, remaining=$remaining" → x=A, y=B, remaining=C D
```

5.8. Iterating over a list: foreach

```
foreach VARLIST LIST ?VARLIST LIST …?  SCRIPT
```

The `foreach` command provides a general purpose means of iterating over a list. The *`VARLIST`* arguments are lists of one or more variable names. The *`LIST`* arguments are the lists to be iterated over. In the simplest case, only a single pair *`VARLIST`*, *`LIST`* is specified and *`VARLIST`* contains a single variable name. For each value in *`LIST`*, the command assigns the value to the variable and executes *`SCRIPT`*.

```
% foreach element {a b c} { puts [string toupper $element] }
→ A
  B
  C
```

When *`VARLIST`* contains more than one variable name, the variables in *`VARLIST`* are assigned consecutive elements from the list at the beginning of each iteration.

```
% set math_english_scores {Mike 90 85 John 85 90 Michelle 90 92}
→ Mike 90 85 John 85 90 Michelle 90 92
% foreach {name math english} $math_english_scores {
    puts "$name got a score of $math in Math and $english in English."
}
→ Mike got a score of 90 in Math and 85 in English.
  John got a score of 85 in Math and 90 in English.
  Michelle got a score of 90 in Math and 92 in English.
```

If the number of elements in *`LIST`* is not a multiple of the number of variable names in the corresponding *`VARLIST`*, the extra variables are assigned the empty string on the last iteration.

In its full form, with multiple pairs of *`VARLIST`* and *`LIST`* specified, `foreach` allows simultaneous iteration over multiple lists. Again, revisiting an earlier example with names and scores stored in separate lists,

```
% set scores(names) {Mike John Michelle}
→ Mike John Michelle
% set scores(math)  {90 85 90}
→ 90 85 90
% foreach name $scores(names) math $scores(math) {
    puts "$name scored $math."
}
→ Mike scored 90.
  John scored 85.
  Michelle scored 90.
```

Note that any number of pairs may be specified and in each pair `VARLIST` may contain more than one variable name. If any list is fewer elements that required, the corresponding variables are assigned the empty strings for the remaining iterations.

As for any looping construct, the `break` (Section 3.11) and `continue` (Section 3.12) commands can be used within a `foreach` script to terminate the loop or skip to the next iteration.

The `foreach` command always returns the empty string as its result.

5.9. Appending elements: lappend

```
lappend VARNAME ?VALUE …?
```

The `lappend` command appends elements to the list stored in a variable whose name is passed to it and presumed to contain a list value. One or more *VALUE* arguments are added to the end of the list contained in *VARNAME*. The resulting list is stored back in the variable and returned as the command result.

```
set greek [list alpha \u03b2]  → alpha β ❶
lappend greek \u03b3 delta     → alpha β γ delta
set greek                      → alpha β γ delta
```

❶ Beta is Unicode character U+03b2

If the variable does not exist, `lappend` will create it as an empty list and then append the remaining arguments. Thus the statements

```
set greek [list alpha beta]
lappend greek alpha beta
```

are equivalent if the variable `greek` did not already exist.

5.10. Inserting elements: linsert

```
linsert LIST INDEX ?VALUE VALUE …?
```

The `linsert` command inserts elements into a list at a specified index and returns the resulting list. The `INDEX` argument must be a single index as the command does not support nested lists. Unlike `lappend` and `lset`, `linsert` operates on a list value `LIST`, not on a variable.

```
set items {a b c d}    → a b c d
linsert $items 1 X Y Z → a X Y Z b c d
```

The `linsert` command treats the index value `end` as the index **after** the last element, not that of the last element. To insert before the last element in the list, specify the index as `end-1`.

```
lindex $items 1        → b
linsert $items 1 X     → a X b c d ❶
lindex $items end      → d
linsert $items end X   → a b c d X ❷
linsert $items end-1 X → a b c X d
```

❶ Item inserted **before** `b`
❷ Item inserted **after** `d`

5.11. Setting element values: lset

```
lset VARNAME ?INDEX …?  VALUE
```

The `lset` command sets the element at specified position in a list stored in a variable to a new value. Like `lappend` (Section 5.9), `lset` operates on variable *VARNAME* that is presumed to contain a list. The element at index *INDEX* in the list is set to *VALUE*. The new list is stored in *VARNAME* and returned as the result of the command.

```
set items [list a {B C} d] → a {B C} d
lset items 2 e             → a {B C} e
```

The command supports nested lists. The multiple indices comprising the path to the nested element may be specified as a single argument or separately.

```
lset items {1 0} X; → a {X C} e
lset items 1 end Y  → a {X Y} e
```

Indices must lie between 0 and the length of the list. In the latter case, the value is appended to the list. Because `lset` treats `end` as index of the last element in the list, appending is done by specifying the index as `end+1`.

```
lset items end f   → a {X Y} f
lset items end+1 g → a {X Y} f g
```


Because `lset` works with nested lists, it can be used to append elements to inner lists, something you cannot do with `lappend`.

5.12. Deleting elements: lremove

```
lremove LIST ?INDEX …?
```

The `lremove` command returns a new list formed by removing zero or more elements at the specified indices from the passed list value.

The indices may be passed in any order or contain duplicates. They specify the elements to be removed, not a path to an element in a nested list.

```
lremove {a b c d e f g} end-1 2 5 → a b d e g
```

The command requires the individual elements' indices to be specified. To remove a contiguous number of elements, use the `ledit` or `lreplace` commands (Section 5.13).

5.13. Replacing elements: lreplace, ledit

```
lreplace LIST FIRST LAST ?VALUE …?
ledit VARNAME FIRST LAST ?VALUE …?
```

The `lreplace` and `ledit` commands replace zero or more elements of a list with zero or more new values.

While `lreplace` operates on the list value *LIST* passed as its first argument, `ledit` operates on the list value stored in the variable *VARNAME* and stores the result back in the variable. In all other respects the commands behave in identical fashion, deleting all elements of the list operand between indices `FIRST` and `LAST` and inserting the additional *VALUE* arguments passed in their place. Both commands return the resulting list value. Neither command supports operating on the inner lists in a nested list.

```
set lst [lreplace {a b c d} 1 2 Y Z] → a Y Z d
ledit lst 0 1 W X                    → W X Z d
set lst                              → W X Z d
```

The commands interpret index `end` as the last element in the list.

```
lreplace {a b c d} end-1 end X Y → a b X Y
```

The number of deletions and additions need not be equal.

```
lreplace {a b c d} 0 2 X Y   → X Y d
lreplace {a b c d} 1 1 X Y Z → a X Y Z c d
```

Moreover, the `VALUE` arguments are optional and if `LAST` is smaller than `FIRST`, no elements are deleted. The commands can therefore be used for pure deletion or insertion.

```
set lst {a b c d}  → a b c d
ledit lst 1 2      → a d
ledit lst 0 -1 X Y → X Y a d
```


Commands that operate on variables are generally more efficient than those operating on values. Therefore `ledit` may be preferable over `linsert` and `lremove` even for pure inserts and deletions if the original value does not need to be preserved. However, as always measure first.

5.14. Counting elements: llength

```
llength LIST
```

The `llength` command returns a count of the number of elements in a list.

```
llength {a b c d e} → 5
```

5.15. Splitting strings into lists: split

```
split STRING ?SEPARATORS?
```

The `split` command divides a string based on a separator character and returns a list containing the resulting substrings. The *`SEPARATORS`* argument is a string that is treated as a **set** of separator characters, not as a single string to be treated as a separator. It defaults to the set of whitespace characters. Thus to break up a paragraph (simplistically) into words based on whitespace,

```
% set paragraph "A sentence.\tAn exclamation! Any questions?"
→ A sentence.     An exclamation! Any questions?
% print_list [split $paragraph]
→ A
  sentence.
  An
  exclamation!
...Additional lines omitted...
```

Or to break it up into sentences based on sentence terminators,

```
% print_list [split $paragraph ".!?"]
→ A sentence
          An exclamation
   Any questions
```

Notice that whitespace is preserved in this last example.

Leading, trailing and consecutive separators will result in empty elements in the resulting list.

```
% split "  one  two  "
→ {} {} one {} two {} {}
```


The `regexp` (Chapter 10) command provides an alternative to `split` for more generalized separator patterns and flexibility. The `splitx` command from the `textutil::split` package in Tcllib[1] is a convenient wrapper around the same.

[1] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

Commonly seen idioms based on `split` include iteration over lines in a file by splitting the content based on newlines and processing strings a character at a time by splitting on an empty string. As an example of the latter,

```
% foreach char [split "string" ""] { puts $char }
→ s
  t
...Additional lines omitted...
```

The `split` command is not an inverse of the `join` command we saw in Section 4.13. In particular, it is not guaranteed that splitting the result of a join using the same character will result in the original list.

```
set l [list a b/c d/e f] → a b/c d/e f
set s [join $l /]        → a/b/c/d/e/f
split $s /               → a b c d e f
```

The presence of the separator character within the list elements will break the inversion.

5.16. Numeric sequences: lseq

```
lseq COUNT ?by STEP?
lseq START count COUNT ? ?by? STEP?
lseq START ?..|to? END ? ?by? STEP?
```

The `lseq` command constructs numeric sequences. It is more efficient in both memory and speed in comparison to construction via an explicit loop.

The `lseq` command is not available in Tcl 8.6 and earlier. Numeric sequences need to be explicitly constructed.

The command can take one of several syntactic forms. The first of these generates a sequence containing `COUNT` elements starting at 0. The optional argument `STEP`, defaulting to `1`, is the difference between consecutive elements and may be negative.

```
lseq 5       → 0 1 2 3 4
lseq 5 by 2  → 0 2 4 6 8
lseq 5 by -1 → 0 -1 -2 -3 -4
```

The second form of the command is similar except that it permits the initial value to be specified. The generated sequence will begin at `START` instead of `0`.

```
lseq -2 count 5      → -2 -1 0 1 2
lseq 2 count 5 by -1 → 2 1 0 -1 -2
lseq 2 count 5 -1    → 2 1 0 -1 -2
```

The `by` keyword is optional as shown above, but the `count` keyword must be specified to distinguish this form from the third described below.

The final form of the command differs from the first two in that it specifies the range covered by its end value as opposed to a count. The sequence is generated by incrementing each prior element by `STEP` until it crosses the `END` value.

```
lseq 1 to 5        → 1 2 3 4 5
lseq 1 .. 5        → 1 2 3 4 5
lseq 1 5           → 1 2 3 4 5
lseq 1 to 5 by 2   → 1 3 5
lseq 1 to 5 by 3   → 1 4
lseq 1 to -5 by -2 → 1 -1 -3 -5
```

The `..` and `to` keywords are synonyms and optional as is `by`. I prefer to use them for readability. Note from the above that inclusion of the `END` value in the sequence depends on whether it differs from `START` by an integral number of steps.

In this third form, if the `STEP` argument is not specified, it defaults to an absolute value of `1` with the sign determined by the relative values of `START` and `END`.

```
lseq -2 to 2 → -2 -1 0 1 2
lseq 2 to -2 → 2 1 0 -1 -2
```

In all the above examples, the numeric values passed to the command were integer literals. However, `lseq` will also accept floating point values as well as expressions in the same syntax as the Tcl `expr` command. For example,

```
lseq 0 2 by 0.5    → 0.0 0.5 1.0 1.5 2.0
set n 5            → 5
lseq {$n-2} {$n+2} → 3 4 5 6 7
```

In the case of floating point sequences, as always you need to be aware the usual issues around floating point computations[2] and rounding that are not specific to `lseq`, or even Tcl, but computers in general.

The `lseq` command can make code more concise. In many cases, `lseq` replaces uses of `for` with more intuitive list based iterations. Below are some examples.

```
set l [lmap i [lseq 1 count 7] { expr {$i*$i} }] → 1 4 9 16 25 36 49 ❶
lremove $l {*}[lseq [llength $l] by 2]           → 4 16 36 ❷
expr {$i in [lseq 1 99 by 2]}                    → 1 ❸
```

❶ Generate a list of first 7 squares.
❷ Remove elements at even indices.
❸ Check for an odd integer within a range.

[2] See https://floating-point-gui.de/.

5.17. Repeating elements: lrepeat

```
lrepeat COUNT ?ELEMENT ELEMENT …?
```

The `lrepeat` command returns a list constructed by repeating its arguments. It is often useful in initialization, for example

```
set download_counters [lrepeat 12 0] → 0 0 0 0 0 0 0 0 0 0 0 0
```

More than one argument may be supplied for repetition.

```
lrepeat 3 a [lrepeat 2 b] → a {b b} a {b b} a {b b}
```

5.18. Concatenating lists: concat

```
concat ?LIST LIST …?
```

The `concat` command returns a new list formed by concatenating zero or more lists.

```
% concat {a b c} {d {e f} g} {} h
→ a b c d {e f} g h
```

Concat preserves any nested list structure; only the outermost lists are merged.

Although `concat` is defined as operating on lists, it does not actually validate that the operands are well-formed lists. In that case, the result may not be a well formed list either. For that reason, many people think of `concat` as a command that operates on strings. However, the Tcl reference describes it as operating on lists so we will stick with that description.

One caveat to be aware of with regard to the use of `concat` with strings is that it will trim any leading or trailing whitespace from each operand. This does not affect list semantics as leading spaces are anyway ignored in the interpretation of strings as lists.

5.19. Mapping list elements: lmap

```
lmap VARLIST LIST ?VARLIST LIST …?  SCRIPT
```

The `lmap` command provides a generalised way to create a new list by mapping elements of a list to new values. As in the case of the `foreach` command, the *`VARLIST`* arguments are lists of one or more variable names. The *`LIST`* arguments are lists of corresponding values. In each iteration of *SCRIPT*, consecutive variables in each *`VARLIST`* are assigned the corresponding values from the *`LIST`* argument following that *`VARLIST`*. The result of the `lmap` command is the list formed from the result of the script in each iteration.

In the simplest form, there is only one *`VARLIST`* argument and it contains only one variable name.

```
lmap n {1 2 3 4 5 6 7 8} {expr {$n*2}} → 2 4 6 8 10 12 14 16
```

As with all loops, the `break` command (Section 3.11) will terminate the iterations. The `continue` command (Section 3.12) continues with the next iteration, skipping the appending of the current iteration result. This is a natural way of filtering a list.

For example, to generate a list containing the squares of all even numbers in a given list

```
% set lst {1 4 5 2 6 3 9 10 12}
→ 1 4 5 2 6 3 9 10 12
% lmap n $lst {
    if {$n % 2 } continue
    expr {$n*$n}
}
→ 16 4 36 100 144
```

The more complex form of `lmap` accepts multiple variables and lists.

```
% lmap {x y} {A B C D} {m n} {1 2 3 4} {
    string cat "$x$m,$y$n"
}
→ A1,B2 C3,D4
```

There need not be the same number of variables in each variable list or the same number of values in each value list. If some value list has fewer values that required, the empty string is assigned to the corresponding variables.

```
% lmap {x y} {A B C D} {n} {1 2 3 4} {
    list $n $x $y
}
→ {1 A B} {2 C D} {3 {} {}} {4 {} {}}
```

5.20. Reversing a list: lreverse

```
lreverse LIST
```

The `lreverse` command returns a list containing elements of the passed list in reverse order.

```
lreverse {a b c d e} → e d c b a
```

The command is often useful in some algorithms where it is more efficient to generate an intermediate result in an order opposite to that desired, and then reverse it for the final result.

5.21. Sorting lists: lsort

```
lsort ?options …? LIST
```

The `lsort` command sorts a list based on some ordering relation.

The default ordering relation compares elements as string values as though `string compare` were used as the comparison function.

```
lsort {b c E g f a d} → E a b c d f g
```

The default sort is case-sensitive and sorts the elements in increasing order.

5.21.1. Comparing elements

The command options that allow selection of the ordering function used for comparison are shown in Table 5.1.

Table 5.1. Lsort comparison options

Option	Description
`-ascii`	Compare using Unicode code-point collation order (default).
`-dictionary`	Compare using "dictionary" order. This differs from `-ascii` in two respects. First, embedded numbers within the strings are compared as integers rather than as character strings. For example, `a100b` will sort after `a9b`. However, a `-` character preceding a number is **not** considered part of the number so in effect all numbers are considered positive. Thus `a-2b` will be considered greater than `a-1b`. The other difference is that case is ignored when comparing strings **except** that if two strings compare as equal when ignoring case, they are then compared in case-sensitive fashion. For example, `abc` compares as less than `Bbc` but greater, **not equal**, to `Abc`. The `-nocase` option is ignored for dictionary comparison.
`-integer`	Treat elements as integers and sort using integer comparisons.
`-real`	Treat elements as floating point numbers and sort using floating point comparisons.
`-command` *COMMAND*	Compare using a caller-defined command. This is passed two arguments and should return a negative integer if the first argument is less than the second, `0` if they are equal, and a positive number if the first is greater than the second.
`-nocase`	Compare ignoring character case. The option is ignored except for the `-ascii` sort mode.

The following examples illustrate the difference between the various options that affect comparisons.

```
set integers {5 10 30}                    → 5 10 30
lsort $integers                           → 10 30 5
lsort -integer $integers                  → 5 10 30
set reals {1.0 0.1e2 5e-2}                → 1.0 0.1e2 5e-2
lsort $reals                              → 0.1e2 1.0 5e-2
lsort -real $reals                        → 5e-2 1.0 0.1e2

set part_numbers {p_100_b P_100_C P_20_B} → p_100_b P_100_C P_20_B
lsort $part_numbers                       → P_100_C P_20_B p_100_b
lsort -ascii $part_numbers                → P_100_C P_20_B p_100_b ❶
lsort -nocase $part_numbers               → p_100_b P_100_C P_20_B
lsort -dictionary $part_numbers           → P_20_B p_100_b P_100_C
```

❶ Same as default

If none of the built-in comparisons are suitable for your purpose, you can use the `-command` option to specify a custom sort ordering. In the example below, we sort the part numbers based on the number of items in stock.

```
proc nstock {part} { return [string length $part] }  ❶
proc compare_stock {s1 s2} { return [expr {[nstock $s1] - [nstock $s2]}] }
lsort -command compare_stock $part_numbers
→ P_20_B p_100_b P_100_C
```

❶ Number in stock happens to match length of part number!

In many cases, instead of sorting using the `-command` option, it is faster to transform the list to a format suitable for sorting using the built-in ordering functions. This is discussed in the Custom sorting[3] page on the Tcler's Wiki[4].

5.21.2. Sort ordering

Specifying the `-increasing` option results in the returned list being sorted with elements in order of lowest to highest. This is the default. The `-decreasing` option can be specified to sort from highest to smallest.

```
set L [list John Paul Ringo George] → John Paul Ringo George
lsort $L                            → George John Paul Ringo
lsort -increasing $L                → George John Paul Ringo
lsort -decreasing $L                → Ringo Paul John George
```

The sort is *stable*, meaning that the ordering of elements that compare as equal will be preserved after the sort. Notice in the examples below that the order in which equal elements are returned is the same as their order in the original list. For instance, `b` and `B` are equal when sorting in case-insensitive mode and their order in the sorted list is the same as the order in the original list.

[3] http://wiki.tcl-lang.org/4021
[4] https://wiki.tcl-lang.org

```
lsort -nocase {b a B} → a b B
lsort -nocase {B a b} → a B b
lsort -real {1 1.0 0} → 0 1 1.0
lsort -real {1.0 1 0} → 0 1.0 1
```

5.21.3. Sorting nested lists with -index

Lists are often used in Tcl for storing structured data similar to records in a database. For example, suppose you need to store records containing students' name and test scores. There are multiple ways this might be done in Tcl, the choice depending on the access patterns and relation with other data.

The most obvious way would be to use a nested list where each inner list contains the person's name and score. This format is often used for results returned from databases.

```
% set students {{Mike 90} {John 85} {Michelle 90} {Ann 92}}
→ {Mike 90} {John 85} {Michelle 90} {Ann 92}
```

Sorting records stored in this manner requires comparisons based on the value of elements in the inner lists. `lsort` provides the `-index` option for this purpose. The option value selects the element of the inner list that is to be used in the sort comparisons. This allows sorting of our student database by either name or test score.

```
% lmap record [lsort -index 0 $students] {lindex $record 0} ❶
→ Ann John Michelle Mike
% lmap record [lsort -index 1 -integer $students] {lindex $record 0} ❷
→ John Mike Michelle Ann
```

❶ Sort by name
❷ Sort by score

In the case of deeply nested lists, you can even pass multiple indices, in which case they will be treated as a path through each nested sub-list, exactly as for `lindex`.

5.21.4. Sorting dictionaries with -stride

Another method of storing records uses a dictionary format which alternates the names and scores.

```
% set student_dict {Mike 90 John 85 Michelle 90 Ann 92}
→ Mike 90 John 85 Michelle 90 Ann 92
```

When structured in this manner the `-stride` option of `lsort` can be used to sort the records. The list is then treated as implicitly consisting of groups of the size specified by the option.

```
% lsort -stride 2 $student_dict
→ Ann 92 John 85 Michelle 90 Mike 90
```

By default the sort comparison element is the first element of each group. So the above fragment will sort based on names. The `-index` option can be used with `-stride` to change this.

```
% lsort -stride 2 -index 1 $student_dict
→ John 85 Mike 90 Michelle 90 Ann 92
```

This now sorts based on the **second** field of the grouping, the score.

Note that `-stride` works equally well for flat lists containing records with more than two fields.

```
% set math_english_scores {Mike 90 85 John 85 90 Michelle 90 92 Ann 92 86}
→ Mike 90 85 John 85 90 Michelle 90 92 Ann 92 86
% lmap {name math english} [lsort -stride 3 -index 1 $math_english_scores] {
    set name ❶
}
→ John Mike Michelle Ann
% lmap {name math english} [lsort -stride 3 -index 2 $math_english_scores] {
    set name ❷
}
→ Mike Ann John Michelle
```

❶ Sort by Math scores
❷ Sort by English scores

5.21.5. Retrieving sorted indices with -indices

One common method of storing data is to maintain a single master list of records which is then sorted multiple ways using different keys (e.g. for display purposes). The `-indices` option instructs the `lsort` command to return the *indices* of the sorted elements instead of the sorted values themselves. We can use this to display our sample data sorted by name or test score without having to worry about consistency maintaining multiple lists.

```
% lmap recnum [lsort -indices -index 0 $students] {
    lindex $students $recnum 0
}
→ Ann John Michelle Mike
% lmap recnum [lsort -indices -index 1 -integer $students] {
    lindex $students $recnum 0
}
→ John Mike Michelle Ann
```

The `-indices` option is also useful when data is stored as parallel lists. For example

```
set scores(names) {Mike John Michelle Ann} → Mike John Michelle Ann
set scores(math)  {90 85 90 92}            → 90 85 90 92
```

Sorting in name order is straightforward but what if we wanted names in order of test scores as we did above? The `-indices` option of `lsort` is useful in this kind of situation where we want to retrieve elements in one list based on a sort order on a different list.

```
lsort -indices -integer $scores(math) → 1 0 2 3
```

We can thus print names in order of test score as follows

```
% lmap recnum [lsort -indices -integer $scores(math)] {
    lindex $scores(names) $recnum
}
→ John Mike Michelle Ann
```

5.21.6. Removing duplicate elements

One final option for `lsort` is `-unique` which returns a sorted list with repeated elements removed.

```
lsort -unique {b a b d a c} → a b c d
```


A common use of the `-unique` option is in the implementation of sets to remove duplicate elements in operations like union.

Note that duplicate elements are those which compare as equal as per the sort options, not just those that identical. Moreover, in case of duplicates, it is the **last** duplicate element from the input list that is preserved. The following example should clarify both these points.

```
lsort -unique {b a B d A c}         → A B a b c d
lsort -nocase -unique {b a B d A c} → A B c d
```

Note how the "last" duplicate is preserved and the impact of the `-nocase` option.

In similar fashion, when the `-indices` option is specified alongside `-unique`, it is the index of the last duplicated element that is included in the returned list.

```
lsort -indices -unique {a c b e d b d} → 0 5 1 6 3
```

In the case of nested lists with the `-unique` option, when the inner elements used for comparison are deemed equal, only the last of the outer elements whose inner elements are equal will be included in the result.

```
lsort -unique -index 0 {{1 a} {3 b} {1 c} {2 d}} → {1 c} {2 d} {3 b}
```

The element `1 a` is not included in the result as the comparison key `1` repeats later.

5.22. Searching lists: lsearch

```
lsearch ?options …?  LIST PATTERN
```

The `lsearch` command searches a list for elements matching specified criteria.

In its simplest form with no options specified, the command returns the index of the first element in *`LIST`* that matches *`PATTERN`*. By default, this matching is done using the rules of `string match` (Section 4.24).

```
lsearch {foo bar jim} b* → 1
```

5.22.1. Search match operators

Table 5.2 lists the options that control the type of matching used.

Table 5.2. Lsearch matching options

Option	Description
`-exact`	The pattern is treated as a literal string with no special characters and compared against list elements for equality. Note that equality does not mean the two strings are identical, for example when the `-integer` option is specified.
`-glob`	Use glob-style matching (the default) as described in Section 4.24.
`-regexp`	Use regular-expression matching as described in Chapter 10.
`-nocase`	Ignore differences in character case.
`-not`	Negate the sense of the match, only including elements **not** matching the pattern.

The options `-exact`, `-glob` and `-regexp` are mutually exclusive. If more than one is specified, the last one takes effect.

Here are a few examples to clarify the various matching types.

```
set l {a a..* a.* ab}  → a a..* a.* ab
lsearch         $l a.* → 1 ❶
lsearch -glob   $l a.* → 1
lsearch -exact  $l a.* → 2
lsearch -regexp $l a.* → 0
lsearch -exact  $l b   → -1
```

❶ Default matching option is `-glob`

When using the `-regexp` option, remember that regular expression matches succeed even if just a substring of the element being compared matches the expression. If you want to match the entire element, constraints like `^` and `$` must be specified.

Any of the above can be combined with the `-nocase` and `-not` options.

```
lsearch                {abc BCD bcd} b*       → 2
lsearch -nocase        {abc BCD bcd} b*       → 1
lsearch -exact         {abc BCD bcd} bcd      → 2
lsearch -nocase -exact {abc BCD bcd} bcd      → 1
lsearch -regexp      {100 abc a10 xyz} {^\d+} → 0
lsearch -not -regexp {100 abc a10 xyz} {^\d+} → 1
```

5.22.2. Search operand types

When exact matching is in effect, the options shown in Table 5.3 modify how strings are interpreted for comparison.

Table 5.3. Lsearch data type options

Option	Description
`-ascii`	Compare using Unicode code-point collation order (default).
`-dictionary`	Use “dictionary” comparison. See the description of the option in Table 5.1. However, as detailed in the Tcl `lsearch` reference page, this only differs from the `-ascii` option if the `-sorted` option is also present.
`-integer`	Compare as integers. Ignored if `-glob` or `-regexp` are in effect.
`-real`	Compare as floating point numbers. Ignored if `-glob` or `-regexp` are in effect.

So, for example, to search for a value in a list of integers irrespective of the integer representation, the `-integer` option is required.

```
lsearch -exact          {0x10 10 16} 16 → 2
lsearch -exact -integer {0x10 10 16} 16 → 0
```

Always include the `-exact` option with the `-integer` or `-real` options. For example,

```
lsearch -integer {0x10 10 16} 16 → 2
```

does not give the expected result because without the `-exact` option the command defaults to `-glob` pattern matching wherein `-integer` has no effect.

5.22.3. Searching nested lists

To locate elements based on values in nested lists, use the `-index` option to indicate the element within each inner list that should be used for comparison.

```
% set students {{Martin 90} {John 85} {Mike 90} {Ann 92}}
→ {Martin 90} {John 85} {Mike 90} {Ann 92}
% lsearch -exact -index 0 $students Mike
→ 2
```

This returns the index in the outermost list that contains the matching element. You can pass the `-subindices` option to get the complete index based path to the matched element. For example, if Mike prefers to be called Michael,

```
% set pos [lsearch -exact -index 0 -subindices $students Mike]
→ 2 0
% lset students $pos Michael
→ {Martin 90} {John 85} {Michael 90} {Ann 92}
```

Note the return value when `-subindices` is specified if no match is found as below.

```
% lsearch -exact -index 0 -subindices $students Albert
→ -1 0
```

5.22.4. Searching grouped lists

The `-stride` option causes the elements within the list to be grouped with an element in each group included in the search. The option's value specifies the size of each group.

By default, the first element of each group is selected for comparison. Alternately, the `-index` option can be passed to select a different element within each group to be compared.

```
set millions [list Tokyo Japan 37 Delhi India 29 Shanghai China 26] → Tokyo Japan
 37 Delhi India 29 Shanghai China 26
lsearch -stride 3 $millions Delhi                                   → 3
lsearch -stride 3 -index 1 $millions China                          → 6
```

Note in the last line above that the index returned is that of the **first** element in the matched group, **not** the matched element.

The `-stride` option is not available in Tcl 8.6 and earlier.

5.22.5. Retrieving all matches

Specify the `-all` option to retrieve indices of all matching elements and not just the first.

```
lsearch -all -index 0      $students M* → 0 2
lsearch -all -index 0 -not $students M* → 1 3
```

5.22.6. Retrieving element values

By default, the `lsearch` command retrieves indices of the matched elements. To retrieve the matched element values instead, specify the `-inline` option.

```
lsearch -index 0              $students M* → 0
lsearch -inline -index 0      $students M* → Martin 90
lsearch -inline -index 0 -all $students M* → {Martin 90} {Michael 90}
```

When used with `-subindices`, `-inline` will only return the matching subelement values, not the whole outer element. So we can get a list of matching names with

```
% lsearch -inline -all -index 0 -subindices $students M*
→ Martin Michael
```

5.22.7. Searching sorted lists

When a list is sorted, `lsearch` can use a more efficient algorithm to locate exact matches. You can indicate that the list is sorted by passing the command the `-sorted` option. This option also implies `-exact` and cannot be used with either the `-regexp` or `-glob` option. The options `-increasing` or `-decreasing` specify the order in which the list is sorted.

```
% lsearch -sorted {ab bc cd} bc
→ 1
% lsearch -all -sorted -integer -decreasing {20 16 0x10 10} 16
→ 1 2
```

Note that the `-sorted` option is primarily a performance feature and does not add any new capabilities to `lsearch`.

Using the `-sorted` option with a list that is not sorted in the expected manner will give erroneous results without raising an error. An example is when the `-nocase` option is used with `lsearch` on a list that was sorted with `lsort` without the `-nocase` option.

The `lsearch` command also provides another very useful option, `-bisect`, when working in conjunction with sorted lists. When specified, `lsearch` returns the index at which the value is found if present (just as if `-bisect` is not specified). If the value is not present, instead of returning `-1`, it will return the position after which the value should be inserted into the list. In the case of lists in increasing order, the returned value is the last index where the element is less than or equal to the searched value. For lists in descending order, it is the last index for which the element is greater than or equal to the search value. If the search value would be placed before the first element in the list or if the list is empty, the command returns `-1`.

```
% lsearch -sorted -integer -bisect -decreasing {20 0x10 16 10} 16
→ 2
% lsearch -sorted -integer -bisect {10 0x10 16} 12
→ 0
```

Note that `-bisect` implies `-sorted` and that it cannot be used in conjunction with the `-all` and `-not` options.

The option is useful in inserting values into a sorted list while maintaining the sort order and not having to resort the list.

```
proc sorted_insert {l val} {
    set pos [lsearch -integer -bisect $l $val]
    if {$pos == -1 || [lindex $l $pos] != $val} {
        return [linsert $l [incr pos] $val]
    } else {
        return $l
    }
}
```

We try it with a value already present and one which is not.

```
% sorted_insert {10 20 30 40} 20
→ 10 20 30 40
% sorted_insert {10 20 30 40} 25
→ 10 20 25 30 40
```

5.22.8. Specifying a start offset

The `-start` option allows control of the index at which a list search begins instead of at `0`. This is useful for conducting multiple searches, each starting where the previous left off.

```
set pos [lsearch {a aby def abz} ab*]     → 1
lsearch -start $pos+1 {a aby def abz} ab* → 3
```

6

Dictionaries

> If anything is guaranteed to annoy a lexicographer, it is the habit of starting a story with a dictionary definition.
>
> — Eric McKean

We will start by defining dictionaries, just to annoy the many lexicographers who will buy this book. A dictionary in Tcl is a data structure that maps each element of a set of strings, called *keys*, to a value. Other languages may refer to these as associative arrays, maps or hash tables.

The keys in a dictionary are always interpreted as strings so, for example, `1` and `0x1` are different keys. The interpretation of values is up to the application which may treat them as strings, integers, lists or even nested dictionaries.

Like lists, dictionaries play many roles in Tcl programming, for example

- as lookup tables to map keys to values
- as C-style records where the key is a field name
- nested tree like structures such as a file system

Correspondingly, Tcl provides commands for a wide variety of operations on dictionaries.

6.1. Dictionary literals

Dictionaries in string form are exactly like lists with an even number of elements that alternate between the key and the associated value.

```
% set colors {red #ff0000 green #00ff00 blue #0000ff}
→ red #ff0000 green #00ff00 blue #0000ff
% dict get $colors red
→ #ff0000
```

Note again, that as explained for list literals, this assigns a *string* to the variable `colors` and it is only when acted on by the `dict` command that it gets interpreted as a dictionary. Thus in

```
% set colors {red #ff0000 green #00ff00 blue}
→ red #ff0000 green #00ff00 blue
% dict get $colors red
Ø missing value to go with key
```

the first statement succeeds as a assignment. The `dict` command fails because the string could not be interpreted as a dictionary as it has an odd number of elements.

As for lists, you do not have to worry about the performance impact as the conversion happens only once as long as the value is operated on with `dict` commands.

6.2. Basic dictionary construction: dict create

```
dict create ?KEY VALUE KEY VALUE …?
```

Like the `list` constructor for lists, more complex dictionaries are best constructed with the `dict create` command. It returns a dictionary containing the specified key/value mappings.

```
% set i 0
→ 0
% set mydict [dict create Key[incr i] Value$i \
                Key[incr i] Value$i \
                Key[incr i] Value$i]
→ Key1 Value1 Key2 Value2 Key3 Value3
```

6.3. Nested dictionaries

Dictionaries may be nested in that the values in a dictionary are also dictionaries. For example, a dictionary containing student data may look like

```
set students {
    A001 {
        Name Jean
        Grades {Physics A Maths A Spanish B}
        Clubs {Chess Photography}
        Age 17
    }
    A002 {
        Name Pedro
        Grades {Maths A Spanish A History B}
        Clubs {Music}
        Age 16
    }
    A003 {
        Name Laxmi
        Age 17
    }
}
```

At the top level, keys are student ids and each value is a nested dictionary containing data for one student. This in turn may contain a nested dictionary, for example the one keyed by `Grades`. Many `dict` subcommands support *key paths*, such as `A001 Grades Physics`, that navigate through dictionaries to access a nested element.

The structure and interpretation of dictionaries is entirely up to an application. Different keys within a dictionary may have values with different structure, some scalars, some lists, some nested dictionaries with arbitrary keys.

6.4. Dictionary and list compatibility

Any list with an even number of elements can be treated as a dictionary with alternating list elements comprising the keys and values. When `ldict` below is accessed using dictionary commands, it will be transparently converted to an internal dictionary form.

```
set ldict [list a 1 b 2 c 3] → a 1 b 2 c 3
dict get $ldict b            → 2
```

The converse is also true as dictionaries can be manipulated using list commands as long as the `-stride` option is used to preserve key and value semantics.

```
% set ldict [lsort -integer -index 1 -decreasing -stride 2 $ldict]
→ c 3 b 2 a 1
```


This last example brings us to a useful property of dictionaries that often comes in handy. Dictionaries are order preserving so that a sequence like the above can be treated both as an ordered list as well as a dictionary. For example, you can use the list form to order data for display while still being to look it up and modify it via indexed dictionary access **without disturbing the order of items**.

6.5. Checking for a key: dict exists

```
dict exists DICTIONARY KEY ?KEY …?
```

The `dict exists` command returns a Boolean true value if the specified key exists in the dictionary and a Boolean false otherwise.

Commands like `dict get` will raise an error on an attempt to retrieve the value for a non-existent key. This command should therefore be used to check for existence.

```
set colors {red #ff0000 green #00ff00 blue #0000ff} → red #ff0000 green #00ff00
 blue #0000ff
dict exists $colors red                             → 1
dict exists $colors yellow                          → 0
```

The command supports nested dictionaries.

```
dict exists $students A001 Grades Physics → 1
dict exists $students A001 Grades Biology → 0
```

6.6. Retrieving the value for a key: dict get|getdef|getwithdefault

```
dict get DICTIONARY ?KEY …?
dict getdef DICTIONARY ?KEY …?  KEY DEFAULT
dict getwithdefault DICTIONARY ?KEY …?  KEY DEFAULT
```

The `dict get` command returns the value corresponding to a key in the dictionary. If no keys are specified, the command returns a list containing the key and value pairs. Specifying a key will return the corresponding value.

```
% set colors {green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff}
→ green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff
% dict get $colors
→ green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff
% dict get $colors red
→ #ff0000
```

The command can also retrieve values from nested dictionaries by specifying multiple keys that define a path through the dictionary.

```
dict get $students A001 Grades Maths → A
```

Note that an attempt to read a key that does not exist in the dictionary will raise a Tcl exception. You can check for the existence of a key before attempting to read it by calling `dict exists` (Section 6.5).

The `dict getdef` and `dict getwithdefault` commands, which are aliases of each other, are similar to `dict get` with the following differences.

- They require at least one *KEY* argument.
- They take an additional *DEFAULT* argument which the command returns as its result if the key does not exist in the dictionary

```
dict get $colors yellow            Ø key "yellow" not known in dictionary
dict getdef $colors yellow #ffffff → #ffffff
```

Like `dict get`, the commands work with nested dictionaries as well.

```
dict getdef $students A001 Grades Art 0 → 0
```

The `dict getdef` and `dict getwithdefault` commands are not available in Tcl 8.6 and earlier. You must explicitly check for the key's existence with `dict exists` before accessing it if it is not guaranteed to be present.

6.7. Enumerating dictionaries: dict keys | values

```
dict keys DICTIONARY ?PATTERN?
dict values DICTIONARY ?PATTERN?
```

The `dict keys` and `dict values` commands returns all keys and values, respectively, that match the passed pattern using the rules of `string match` (Section 4.24). If *`PATTERN`* is not passed, the commands return all keys or values.

```
dict keys $colors        → green red blue magenta
dict keys $colors *r*    → green red
dict values $colors      → {#00ff00} #ff0000 #0000ff #ff00ff
dict values $colors #ff* → {#ff0000} #ff00ff
```

The values are matched as strings no matter whether they are integers, nested lists etc.

Not relevant to the current discussion, but you may wonder why the first element in the string representation of a list is enclosed in braces if it begins with a `#` as was the case above. The short answer is that when a command (or command prefix) is constructed in list form, enclosing the first word of the command in braces prevents it from being mistakenly parsed as a comment. For a fuller explanation see TIP 407[1]. Note this does **not** change the first element value! So for example retrieving the first element above

```
lindex [dict values $colors #ff*] 0 → #ff0000
```

correctly retrieves the value `#ff0000`.

6.8. Setting values with dict set

```
dict set DICTVAR KEY ?KEY …?  VALUE
```

The `dict set` command operates on variable *`DICTVAR`*, presumed to contain a dictionary. If the specified key exists in the dictionary stored in *`DICTVAR`*, its value is replaced with the new value. If the key does not exist, it is added to the dictionary along with the associated value. The resulting dictionary is returned by the command as well as stored back into *`DICTVAR`*.

```
set mydict [dict create a 1 b 2 c 3] → a 1 b 2 c 3
dict set mydict a 10                 → a 10 b 2 c 3 ❶
dict set mydict x 100                → a 10 b 2 c 3 x 100 ❷
set mydict                           → a 10 b 2 c 3 x 100
```

❶ Existing key
❷ New key

[1] http://www.tcl-lang.org/cgi-bin/tct/tip/407

If multiple *KEY* arguments are passed, they are treated as a path through a nested dictionary. If the path does not exist, the command will create any missing keys as necessary.

For example, to add a new student

```
dict set students A004 {
    Name Mark
    Grades {
        Physics A
    }
}
```

To update an existing student, we could do the update one leaf element at a time,

```
dict set students A003 Grades Physics B
dict set students A003 Grades English A
```

or with the entire nested dictionary

```
dict set students A003 Grades {Physics B English A}
```

The above two sequences are equivalent only because the key `Grades` did not previously exist for student id `A003`. If the key did in fact already exist, the first form that updated a leaf at a time would **add** the new keys under the existing `Grades` key. The second form on the other hand would **replace** the existing contents of `Grades`.

6.9. Removing dictionary elements: dict unset|remove

```
dict unset DICTVAR KEY ?KEY …?
dict remove DICTIONARY ?KEY …?
```

The `dict unset` command removes individual elements from a dictionary stored in a variable, stores it back in the variable and returns it as the command result.

Thus if Laxmi leaves the school, we can forget about her existence.

```
dict unset students A003
```

As for `dict set`, a key path can be specified to remove any nested element. If Jean has not actually taken Spanish and her grade was erroneously given we can correct the error.

```
dict unset students A001 Grades Spanish
```

It is not an error if the last key on the key path (`Spanish` in our example) is missing. However keys other than the last must exist else the command will raise an exception.

The `dict remove` command is an alternative means of removing elements from a dictionary. It differs from `dict unset` in two respects:

- It operates on a dictionary **value** whereas `dict unset` operates on a **variable**.
- Multiple key arguments refer to the top level keys to be removed, **not** a single key path to a nested element.

Keys that do not exist in the dictionary are ignored and do not raise an error. The command returns the dictionary resulting from removal of the elements.

```
% set mydict {a 1 b 2 c 3 d 4}
→ a 1 b 2 c 3 d 4
% dict remove $mydict a c
→ b 2 d 4
```

6.10. Appending to string values: dict append

```
dict append DICTVAR KEY ?STRING …?
```

The `dict append` command concatenates the supplied *`STRING`* arguments, appending the result to the dictionary element corresponding to the specified key, creating it if necessary. The resulting dictionary is stored back in *`DICTVAR`* and returned as the command result.

```
set mydict [dict create keyA A] → keyA A
dict append mydict keyA BC      → keyA ABC ❶
dict append mydict keyW W XYZ   → keyA ABC keyW WXYZ ❷
set mydict                      → keyA ABC keyW WXYZ
```

❶ Append to an existing key
❷ Create a new key and append multiple strings

Only a single level of keys can be specified so nested dictionaries cannot be directly modified.

One way around this limitation is to structure the two-level dictionary as a one-level dictionary stored in an array (Section 6.22).

6.11. Appending list elements to values: dict lappend

```
dict lappend DICTVAR KEY ?VALUE …?
```

The `dict lappend` command retrieves the value, which should be interpretable as a list, currently associated with the key *`KEY`* in the dictionary stored in *`DICTVAR`*, appends the given elements to it in the same manner as `lappend` (Section 5.9). The resulting dictionary is stored back in *`DICTVAR`* and returned as the command result.

Like `dict append`, `dict lappend` does not support nested dictionaries. Thus to update our `students` dictionary to reflect that Pedro has joined the Athletics club we have to extract the nested dictionary, modify it and write it back.

```
set pedro [dict get $students A002]
dict lappend pedro Clubs Athletics
dict set students A002 $pedro
```

6.12. Incrementing dictionary values: dict incr

```
dict incr DICTVAR KEY ?INCREMENT?
```

The `dict incr` command increments the value of the element in the dictionary contained in *DICTVAR* with key *KEY* by *INCREMENT* which defaults to 1. The resulting dictionary is stored back in *DICTVAR*. If the key did not exist in the dictionary, it is created with an initial value of 0 before being incremented by the specified amount.

Below is a simple example of maintaining word counts using a dictionary.

```
foreach word {Do what you can, ignore what you can't.} {
    dict incr word_counts $word
}
puts $word_counts
→ Do 1 what 2 you 2 can, 1 ignore 1 can't. 1
```

6.13. Replacing multiple values: dict replace

```
dict replace DICTIONARY ?KEY VALUE …?
```

The `dict replace` command returns a new dictionary formed by replacing the values for specified keys in *`DICTIONARY`*.

The command creates new entries for keys that do not exist.

```
% set mydict {a 1 b 2 c 3 d 4}
→ a 1 b 2 c 3 d 4
% dict replace $mydict a 10 c 30 X 50
→ a 10 b 2 c 30 d 4 X 50
```

6.14. Combining dictionaries: dict merge

```
dict merge ?DICTIONARY DICTIONARY …?
```

The `dict merge` command creates a new dictionary by combining the content of multiple existing dictionaries. The returned dictionary will include the union of the keys in the passed dictionaries. The corresponding value will be the one associated with the key in the last dictionary argument containing that key.

Consider for instance a word processor or Web browser where the appearance of text depends on settings specified at the page, paragraph or individual text span levels. These options can be stored in dictionaries for each level and combined for displaying text using `dict merge`.

```
set page_settings {font-family Helvetica background white foreground black}
set para_settings {font-family Arial}
set link_settings {foreground blue font-style underlined}
set settings [dict merge $page_settings $para_settings $link_settings]
→ font-family Arial background white foreground blue font-style underlined
```

Note the order of merge so the settings in `link_settings` take precedence.

6.15. Iterating over dictionaries: dict for

```
dict for {KEYVAR VALUEVAR} DICTIONARY SCRIPT
```

The `dict for` command iterates over every entry in a dictionary.

The command executes *`SCRIPT`* for every entry in the dictionary **in the order that the keys were inserted into it**. In each iteration, the variables named *`KEYVAR`* and *`VALUEVAR`* are assigned the key and the value of next entry in the dictionary. Like other Tcl commands that loop, the iteration can be terminated by a `break` command (Section 3.11) before all entries are processed. Likewise, a `continue` command (Section 3.12) will skip the rest of the script but continue on with the next entry.

```
dict for {color rgb} $colors {
    puts "The RGB value for $color is $rgb."
}
→ The RGB value for green is #00ff00.
  The RGB value for red is #ff0000.
...Additional lines omitted...
```

6.16. Mapping values: dict map

```
dict map {KEYVAR VALUEVAR} DICTIONARY SCRIPT
```

The `dict map` command returns a new dictionary formed by mapping each value in the dictionary to a new value returned by a script.

The command executes *`SCRIPT`* for each entry in the dictionary assigning the key and value to variables *`KEYVAR`* and *`VALUEVAR`*. On normal completion of each iteration, a new entry is added to the result dictionary with the key and value being the current content of *`KEYVAR`* and *`VALUEVAR`*. The iteration is terminated by a `break` (Section 3.11) while a `continue` (Section 3.12) is continues with the next iteration without changing the result dictionary for the current iteration.

As a simplistic example, to convert colors to 8-bit grey scale by averaging RGB values,

```
set grey_scale [dict map {color rgb} $colors {
    regexp {^#(..)(..)(..)$} $rgb -> r g b ❶
    format "#%x" [expr {("0x$r" + "0x$g" + "0x$b")/3}]
}]
→ green #55 red #55 blue #55 magenta #aa
```

❶ Split into red, green, blue components

If the variable containing the key is modified, the new dictionary will contain a new key corresponding to the modified content of the variable.

```
set grey_scale [dict map {color rgb} $colors {
    regexp {^#(..)(..)(..)$} $rgb -> r g b
    set color "greyscale_$color"
    format "#%x" [expr {("0x$r" + "0x$g" + "0x$b")/3}]
}]
→ greyscale_green #55 greyscale_red #55 greyscale_blue #55 greyscale_magenta #aa
```

6.17. Filtering dictionaries: dict filter

```
dict filter DICTIONARY key ?PATTERN …?
dict filter DICTIONARY value ?PATTERN …?
dict filter DICTIONARY script {KEYVAR VALUEVAR} SCRIPT
```

The `dict filter` command returns a new dictionary containing entries from an existing dictionary that meet specified matching criteria.

In the first form, the command will return a new dictionary that contains every entry whose key matches at least one of the *`PATTERN`* arguments using `string match` rules (Section 4.24). If no patterns are specified, an empty dictionary is returned.

For example, to filter our display settings example from Section 6.14 to only get font settings,

```
dict filter $settings key font* → font-family Arial font-style underlined
```

The second form of the command is similar except that instead of matching against the key for an entry, it matches against the value. To exclude all colors that have a green component,

```
dict filter $colors value #??00?? → red #ff0000 blue #0000ff magenta #ff00ff
```

The final form of `dict filter` is the most flexible. It executes *`SCRIPT`* for every entry in the dictionary. On each iteration, the key and value are assigned to the variables named *`KEYVAR`* and *`VALUEVAR`* respectively. The entry is included in the dictionary only if the iteration returns a Boolean true value. The `break` command (Section 3.11) terminates the iteration. A `continue` command (Section 3.12) is treated the same as a Boolean false return from execution of the script.

The following returns a dictionary containing colors with either a green or a blue component.

```
dict filter $colors script {color rgb} {
    expr {![string match #??0000 $rgb]}
}
→ green #00ff00 blue #0000ff magenta #ff00ff
```

6.18. Shadowing dictionaries with local variables: dict update

```
dict update DICTVAR KEY VARNAME ?KEY VARNAME …?  SCRIPT
```

Earlier we saw commands like `dict lappend` that directly update a dictionary entry. In the general case though, updating an entry requires retrieving it with `dict get`, modifying it, and then storing it back which is what we had to do when Pedro joined the Athletics club. The `dict update` command encapsulates this sequence of retrieval, modification and writing back into the dictionary.

The command looks up each specified key in the dictionary contained in *`DICTVAR`* and assigns its value to corresponding variable *`VARNAME`*. If a specified key is not present, the corresponding variable remains undefined unless it was already defined in the scope. The command then executes the specified script on the completion of which the values in each of *`VARNAME`* are assigned back to the corresponding key in the dictionary. The modified dictionary is stored back in *`DICTVAR`*. The return value from the command is the return value of the last statement executed in *`SCRIPT`*. The example below illustrates the possibilities.

```
% set xvar X
→ X
% set mydict {a 1 b 2 c 3}
→ a 1 b 2 c 3
% dict update mydict a avar c cvar d dvar x xvar {
    incr avar 10 ❶
    unset cvar ❷
    set dvar [dict get $mydict a] ❸
}
→ 1
% puts $mydict ❹
→ a 11 b 2 d 1
% info exists xvar ❺
→ 0
```

❶ Will change value associated with key `a`
❷ Will result in key `c` being removed
❸ Will add a new key `d` with *old* value of key `a` as `mydict` is still unchanged at this point
❹ Variable `mydict` is updated *after* completion of script
❺ Since key `x` did not exist in dictionary, previously existing variable `xvar` is unset

The value of `mydict` is updated even when the script raises an error exception.

The command can also be used with nested dictionaries. Consider an example similar to one mentioned earlier — here we want to update the dictionary to reflect the fact that Jean joined the archery club on her birthday. Either of the following would do the job, using `dict update` only at the first level as below:

```
set student_id A001
dict update students $student_id student {
    dict lappend student Clubs Archery
    dict incr student Age
}
dict get $students A001
→ Name Jean Grades {Physics A Maths A} Clubs {Chess Photography Archery} Age 18
```

Or using `dict update` at both levels in nested fashion:

```
set student_id A001
dict update students $student_id student {
    dict update student Age age Clubs clubs {
        lappend clubs Archery
        incr age
    }
}
dict get $students A001
→ Name Jean Grades {Physics A Maths A} Clubs {Chess Photography Archery} Age 18
```

The `dict update` command is really most useful when the update is more complex than the simplistic examples shown here, particularly when the update involves more than one key from the dictionary.

6.19. Shadowing nested dictionaries: dict with

```
dict with DICTVAR ?KEY …?  SCRIPT
```

The `dict with` command is similar to the `dict update` command in that it executes a script with variables that shadow dictionary entries and then writes them back into the dictionary.

If no *`KEY`* arguments are specified, the command executes *`SCRIPT`* after assigning the values in the dictionary in variable *`DICTVAR`* to variables of the same name as the corresponding dictionary keys. When the script completes, any changes made to those variables are written back to the corresponding keys in the dictionary contained in *DICTVAR*. The result of the script execution is returned as the result of the command.

The value of `mydict` is updated even when the script raises an error exception.

The command differs from `dict update` in the following respects:

- unlike `dict update`, dictionary keys are mapped to variables of the same name with no provision to map to variables of a different name. This requires some care to prevent conflict between dictionary keys and existing variables having the same name.
- `dict with` makes it easy to deal with nested dictionaries. If one or more *`KEY`* arguments are specified, instead of shadowing the top level keys of the dictionary with variables, the command shadows the nested dictionary identified by the specified key path *`KEY …`* .
- `dict with` can only update existing keys, not create new ones.

Here is a basic example of `dict with`.

```
set mydict {a 1 b 2 c 3}
dict with mydict {
    incr a $a
    incr b $b
}
puts $mydict
→ a 2 b 4 c 3
```

Here is an example for nested dictionaries that updates Jean's grades.

```
dict with students $student_id Grades {
    set Physics A-
    set Maths B
}
puts [dict get $students $student_id]
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery} Age 18
```


Since retrieving key values from dictionaries with `dict get` can be tedious, a trick you will often see employed is to pass an empty script to `dict with` for the sole purpose of bringing the keys into local scope and access them like any other variables. So for instance, instead of

```
% puts "RGB are [dict get $colors red], \
              [dict get $colors blue], \
              [dict get $colors green]"
→ RGB are #ff0000,  #0000ff,  #00ff00
```

we can do this

```
% dict with colors {}
% puts "RGB values are $red, $blue, $green"
→ RGB values are #ff0000, #0000ff, #00ff00
```

which can be convenient in longer scripts.

A cautionary note

It is important to keep in mind that the variables affected by `dict with` are dependent on the contents of the dictionary. Unexpected behaviour can result if a dictionary key happens to be the same as the name of an unrelated variable which will get overwritten. One way of minimizing this possibility is to adopt a convention where dictionary keys are syntactically different from variable names; for example, making them all upper case or starting with an upper case letter.

Under some circumstances this overwriting of variables can even be a security risk. For instance, some Tcl web servers will return received URL parameters as a dictionary mapping the client supplied parameter to a value. Passing this dictionary to `dict with` will allow the client to overwrite any variable, even global ones, with their own chosen values. In general, avoid using `dict with` and `dict update` with dictionaries constructed from arbitrary input values.

6.20. Count of entries: dict size

```
dict size DICTIONARY
```

The `dict size` command returns the number of entries in a dictionary.

```
dict size $colors → 4
```

6.21. Dictionary statistics: dict info

The `dict info` command returns a human readable string that provides some information about the internal structure of the dictionary.

```
% dict info $colors
→ 4 entries in table, 4 buckets
  number of buckets with 0 entries: 1
  number of buckets with 1 entries: 2
  number of buckets with 2 entries: 1
  number of buckets with 3 entries: 0
...Additional lines omitted...
```

The command is primarily intended for debugging and performance related analysis.

6.22. Dictionaries versus arrays

Because they both provide facilities for mapping keys to values, there is often confusion regarding the differences between arrays and dictionaries and the circumstances in which each is to be preferred. The table below highlights these differences.

Table 6.1. Differences between tables and arrays

Arrays	Dictionaries
Arrays are collections of *variables*. For example, `myarray($key)` is a variable and can be accessed as `$myarray($key)`.	Dictionaries are collections of *values* where individual values cannot be accessed as variables. They must be accessed as `dict get $mydict $key`.
Because they are variables, it is possible to set variable traces on individual array elements.	Variable traces can only be set on the entire dictionary, not individual entries in the dictionary.
Arrays are not values and cannot be directly passed to procedures without using additional mechanisms such as `upvar` (Section 14.1.4).	Dictionaries are values and can be passed into procedures like any other value.
Arrays cannot be nested as they are not values.	Dictionaries can be nested because they can hold any values including other dictionaries.
Arrays are *unordered* collections and hence the order in which elements are accessed in operations like iteration is not guaranteed.	In iteration and similar operations, dictionary entries are always processed in the order in which the keys for the entries were created.

Dictionaries contain values and therefore cannot contain arrays. On the other hand, dictionaries are values and therefore can be contained in arrays. This fact is often useful in cases where there are two levels of keys. Structuring the data such that the first level of keys are stored in arrays and the second level in dictionaries can make certain accesses more convenient.

For example consider our `students` dictionary data store and convert it to an array form where the array elements indexed by student id will hold a dictionary containing the "record" for that student. This is easy enough to do.

```
% array set student_array $students
% puts $student_array(A001)
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery} Age 18
```

Now to modify a student's record, we can directly use dictionary in-place commands like `dict lappend`. In our earlier examples, we could not use these directly because they do not support nested dictionaries causing us to instead do a read/modify/write cycle instead.

```
% dict lappend student_array(A001) Clubs Gymnastics
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery Gymnas...
% dict get $student_array(A001) Clubs
→ Chess Photography Archery Gymnastics
```

7

Numerics

> All which is beautiful and noble is the result of reason and calculation.
>
> — Charles Baudelaire

Although Tcl, like most dynamic languages, is not intended for heavy numeric computation, it provides a full set of operators and functions that should more than suffice for general purpose computing.

7.1. Types and representations

Tcl supports operations on boolean, integer and floating point values. In this section we go over these in terms of their representation, acceptable values and conversions.

> **Internal representation of numbers**
>
> At the scripting level, there is no reason to be concerned with the internal representations. However, folks might wonder whether Tcl will convert numerics and strings back and forth on every arithmetic operation and how that might affect performance. The answer is that internally Tcl will keep numbers in the usual native form for the machine. It is only when they are used as strings, for printing for example, are the string representations generated.

7.1.1. The boolean type

Tcl accepts the following values as booleans:

- a boolean false is a numeric value of `0` or the strings `false`, `no`, `off`
- a boolean true is any non-0 numeric value and the strings `true`, `yes`, `on`

The string values are case-insensitive and **unique** abbreviations are also acceptable.

```
if {1} {puts true!}                     → true!
if {tRue} {puts true!}                  → true! ❶
if {fal} {puts true!} else {puts false!} → false! ❷
```

❶ Case insensitive
❷ Abbreviated from `false`.

In computed boolean expressions, Tcl returns boolean true as `1` and false as `0`.

7.1.2. The integer types

Tcl supports integers of arbitrary size so just in case you wanted to calculate the number of stars in the universe, you could.

```
set ngalaxies 10000000000                    → 10000000000
set stars_per 100000000000                   → 100000000000
set nstars [expr {$ngalaxies * $stars_per}] → 1000000000000000000000
```

The advantage over floating point representation is that you don't lose precision and there is no theoretical limit to the number of digits.

Tcl accepts several representations for integers. `123` may be represented in

- base-10 as decimal digits, optionally prefixed with `0d` or `0D` — `123` or `0d123`
- base-16 as hexdecimal digits, prefixed by `0x` or `0X` — `0x7b`.
- base-8 as octal digits, prefixed with `0o` or `0O` — `0o173`.
- base-2 as ones and zeroes, prefixed with `0b` or `0B` — `0b1111011`.

Tcl 8.6 treated **any** numeric string beginning with a `0` character, such as `0777`, as a number in octal representation. This is no longer the case in Tcl 9. Avoid such use, as aside from Tcl 9 compatibility, it can lead to unexpected results even in Tcl 8.

All these forms may be preceded by a `-` character to represent negative integers. Additionally underscores may be used to separate characters for readability.

```
set a 1_0_1          → 1_0_1
expr {$a + 100_000} → 100101
```

7.1.3. The floating point type

Floating point, or real number, values are represented internally as the C language `double` type. The string representation takes the form of

- an optional `-` or `+` sign
- followed by a string of decimal digits containing at most one decimal point
- optionally followed by the exponent which consists of an `e` or `E` character followed by an optional sign, and then a string of decimal digits.

As for integer representations, underscores may be used as separators for better readability. The values `1`, `1.0`, `-1.0e100`, `10E-100`, `1_000_000.001_001` are all valid floating point values.

The positive and negative infinities are represented by `Inf` and `-Inf` respectively, and behave as you would expect in most calculations.

```
expr 1.0 / 0.0 → Inf
expr Inf * 2   → Inf
```

The other special case related to floating point representation is the "not a number" value represented by `NaN`.

```
tcl::mathfunc::sqrt -1 → -NaN
```

We can confirm that both these are treated as floating point values.

```
string is double Inf → 1
string is double NaN → 1
```

7.1.3.1. Floating point classification: fpclassify

```
fpclassify FPVALUE
```

The `fpclassify` command returns the class of a floating point value. Possible classes are shown in Table 7.1.

Table 7.1. Floating point classes

Class	Description
`infinite`	Floating point infinity
`nan`	Not a Number
`normal`	Any value not belonging to the other classes.
`subnormal`	Result of a gradual underflow.
`zero`	Floating point zero.

The `fpclassify` command is not available in Tcl 8.6 and earlier.

7.1.4. Validation of types

The `string is` command can be used to validate that a passed value is an acceptable string representation for a type.

```
string is integer 123        → 1
string is integer 1_2_3      → 1
string is integer abc        → 0
string is integer 1.1        → 0
string is integer ""         → 1 ❶
string is integer -strict "" → 0
```

❶ Note an empty string is acceptable as any type unless `-strict` is specified.

Other numeric types such as booleans, doubles, etc. can be validated in similar fashion. See Section 4.23 for details.

7.1.5. Number conversions

7.1.5.1. Converting between strings and numbers

A number may have multiple string representations. For example, the integer 10 may be represented as `10`, `0xa`, `0x0A` and so on. The same also applies to floating point numbers.

Tcl commands operating on numbers will accept any of these representations as a valid number. If however, if you want to validate that the string is a number in a **specific** form, you can use the `scan` (Section 4.21) or `regexp` (Chapter 10) commands.

Conversely, when converting the numeric result of a computation into a string for display purposes or other reasons, Tcl will generate its "natural" representation. For integers, this is in the form of a string of decimal digits. For floating point numbers, Tcl generates a string that contains the minimal number of digits required to distinguish the number from its nearest floating point neighbours[1].

You can use the `format` command (Section 4.20) to generate a specific string representation.

In Tcl 8, generation of string representations for floating point numbers could be controlled by the `tcl_precision` global. Its use was deprecated due to numerous pitfalls and is no longer available in Tcl 9.

7.1.5.2. Converting between numeric types

For situations where you want to explicitly control the numeric type, Tcl provides a set of commands to "cast" a value to the desired numeric type. These commands lie within the `::tcl::mathfunc` namespace.

The `int`, `wide`, `entier` commands return the integer portion of their argument. The `int` and `entier` are synonyms and do not truncate the resulting integer while `wide` truncates to the low 64 bits. Similarly, the `bool` and `double` commands convert their operand to a boolean and floating point value respectively.

In Tcl 8.6, the `int` command truncates to 32-bits. In Tcl 9, if you want to truncate integer values, explicitly use `expr` to mask off the high bits.

```
tcl::mathfunc::int 2.5                          → 2
tcl::mathfunc::bool 2.5                         → 1
tcl::mathfunc::int 1.4e23                       → 140000000000000008388608
tcl::mathfunc::wide 1.4e23                      → 7659224618221174784 ❶
tcl::mathfunc::double 140000000000000000000000 → 1.4e+23
```

❶ Low 64 bits

[1] If you do not understand this statement, it arises from the fact that floating point representations in computer arithmetic are inexact approximations. See http://blog.reverberate.org/2014/09/what-every-computer-programmer-should.html for an explanation.

As we will see shortly, the commands in the `tcl::mathfunc` namespace can be used as functions in Tcl expressions evaluated by `expr` (Section 7.2.2) so the above could also be written as

```
expr { int(2.5) } → 2
```

7.2. Mathematical operations

Mathematical operations in Tcl can be executed in one of two ways:

- Each arithmetic operation is implemented as a command in the `::tcl::mathop` namespace.
- The `expr` (Section 7.2.2) command implements in-fix expressions as found in other languages.

7.2.1. The tcl::mathop commands

Commands corresponding to the common arithmetic operations are located in the `::tcl::mathop` namespace. For example, you can add three numbers as

```
% tcl::mathop::+ 1 2 3
→ 6
```

We have not looked at namespaces yet and will do so in Chapter 16. For now you can invoke the commands as above, or to reduce the typing involved, run the following command

```
% namespace path ::tcl::mathop
```

You can then simply type

```
% + 1 2 3
→ 6
```

As reflected by the name, the `tcl::mathop` namespace primarily contains commands related to mathematical operations. However, it also includes some operators that work with non-numeric operands, returning boolean values that can then be used in expressions.

The `tcl::mathop` commands can be grouped into the following categories:

- Arithmetic operators
- Comparison operators
- String operators
- List operators
- Bitwise operators

7.2.1.1. Arithmetic operator commands

Table 7.2 lists all the commands dealing with arithmetic operations and the operand types to which they apply.

Table 7.2. Arithmetic operators

Operator	Result
`!` *BOOL*	Complement of the boolean argument.
`+` ?*NUM* ...?	Sum of all arguments.
`*` ?*NUM* ...?	Product of all arguments.
`-` *NUM* ?*NUM*...?	Subtraction or negation. See examples below.
`/` *NUM* ?*NUM* ...?	Division or floating point reciprocal. See examples below.
`%` *INT INT*	Integral remainder from division of the first argument by the second. The result's sign will be the same as that of the second operand.
`**` ?*NUM* ...?	First operand successively raised to the power specified by subsequent operands. Evaluated right to left (see example).

The following examples illustrate some specific cases of these operators.

```
tcl::mathop::! 2            → 0 ❶
tcl::mathop::! false        → 1
tcl::mathop::- 10           → -10 ❷
tcl::mathop::- 10 3 2 1     → 4 ❸
tcl::mathop::/ 2            → 0.5 ❹
tcl::mathop::/ 0            → Inf
tcl::mathop::/ 9 2 2.0      → 2.0 ❺
tcl::mathop::/ 9 2.0 2      → 2.25 ❻
tcl::mathop::% 13 -3        → -2 ❼
tcl::mathop::* [/ 13 -3] -3 → 15
tcl::mathop::- 13 [% 13 -3] → 15
tcl::mathop::** 2 3 4       → 2417851639229258349412352 ❽
tcl::mathop::** 4 0.5       → 2.0 ❾
```

❶ Boolean results are always `0` or `1`.
❷ Single argument is negation.
❸ Multiple arguments is subtraction of subsequent operands from the first.
❹ Single argument is reciprocal with result always being a floating point number.
❺ Multiple arguments is successive division of first operand by subsequent ones.
❻ First floating point argument forces floating point computation.
❼ Integral remainder has same sign as second operand so following two results are equal.
❽ Computed as 2**(3**4), not (2**3)**4.
❾ Floating point result if any argument is floating point.

7.2.1.2. Comparison operator commands

The second set of operators is used for comparisons. The operands are compared pair-wise. If each operand in the pair is numeric, they are compared as numbers. If either is not numeric, they are compared as strings.

The comparison operators are shown in Table 7.3.

Table 7.3. Comparison operators

Operator	Result
`==` ?*ARG* ...?	`1` if every argument equals its neighbours, else `0`. `tcl::mathop::== 0xa 10.0 0o12 → 1`
`!=` *ARG ARG*	`1` if the two arguments are not equal, else `0`.
`<` ?*ARG* ...?	`1` if every argument is less than the next, else `0`.

`<=` ?*ARG* ...?	`1` if every argument is less than or equal to the next, else `0`.
	`tcl::mathop::<= 10 20 30 40 → 1`
	`tcl::mathop::<= 10 20 40 30 → 0`
	`set val 10 → 10`
	`tcl::mathop::<= 0 $val 20 → 1` ❶
	❶ Check if a value is within a range

`>` ?*ARG* ...?	Returns `1` if every argument is greater than the next, else `0`.
`>=` ?*ARG* ...?	Returns `1` if every argument is greater than or equal to the next, else `0`.

We reiterate that numeric comparisons are used if both operands can be interpreted as numbers. If not, they are compared as strings. This means you have to be careful where you really want string comparisons and there is a chance operands might appear to be numbers (e.g. ZIP codes). In this case use the string operators (Table 7.4) or the `string compare` command (Section 4.22) instead.

Moreover, when more than two operands are specified, each comparison is done in isolation so that one comparison may be numeric and another string. For example,

```
tcl::mathop::< " a" 12 2 → 0
```

The first comparison is done as a string and evaluates to true. The second comparison would also evaluate to true were if done as a string comparison. But because both `12` and `2` can be interpreted as numbers, it is treated as a numeric comparison and thus returns false. Again, this illustrates the need to be careful about the operands and whether they can be interpreted as numbers in such cases.

7.2.1.3. String operator commands

Operators that compare strings are shown in Table 7.4 and are equivalent to the `string equal` and `string compare` commands (Section 4.22). All operators are case-sensitive.

Table 7.4. String comparison operators

Operator	Result
`eq` ?*ARG* ...?	`1` if every argument equals its neighbours, else 0.
`ne` *ARG ARG*	`1` if the two arguments are not equal, else `0`
`lt` ?*ARG* ...?	`1` if every argument is less than the next, else `0`.
`le` ?*ARG* ...?	`1` if every argument is less than or equal to the next, else `0`.
`gt` ?*ARG* ...?	`1` if every argument is greater than the next, else `0`.
`ge` ?*ARG* ...?	`1` if every argument is greater than or equal to the next, else `0`.

The following checks if a list is in strictly increasing order.

```
tcl::mathop::lt apple banana orange → 1
tcl::mathop::lt apple orange banana → 0
```

The difference compared to `==`, `!=` et al is that the string operators always compare the operands as strings. This is illustrated by the following.

```
== 0xa 10  → 1
eq 0xa 10  → 0
!= NaN NaN → 1
ne NaN NaN → 0
```

The `NaN` example may surprise you. The `NaN` value is a valid floating point value and thus the `!=` operator treats it as such. The “specialness” of this *Not a Number* value is that it will compare as being unequal to all values, **even itself**! As a string, it compares equal to itself.

7.2.1.4. List operator commands

The commands in Table 7.5 check for list containment. The elements are treated purely as strings and the comparison is case-sensitive.

Table 7.5. List membership operators

Operator	Result
`in` *ARG LIST*	`1` if *ARG* is an element in *LIST*, else 0.
`ni` *ARG LIST*	`1` if *ARG* is not an element in *LIST*, else 0.

```
in apple {apple banana orange} → 1
ni apple {apple banana orange} → 0
```

7.2.1.5. Bit-wise operator commands

The next set of operators deal with bit-wise operations and are shown in Table 7.6. The operands to these must be integers (of any width).

Table 7.6. Bit operators

Operator	Result
`~` *INT*	Bit-wise negation of *`INT`*.
`&` ?*INT* ...?	Bit-wise AND of all arguments.
`|` ?*INT* ...?	Bit-wise OR of all arguments.
`^` ?*INT* ...?	Bit-wise XOR of all arguments.
`<<` *INT SHIFT*	*`INT`* shifted left by *`SHIFT`* bits.
`>>` *INT SHIFT*	*`INT`* shifted right by *`SHIFT`* bits. Sign bit is propagated.

```
proc bitdemo {op args} {
    format 0b%b [$op {*}$args] ❶
}                              → (empty)
bitdemo ~ 0b1100               → 0b11111111111111111111111111110011
bitdemo & 0b1100 0b1010        → 0b1000
bitdemo | 0b1100 0b1010        → 0b1110
bitdemo ^ 0b1100 0b0101        → 0b1001
bitdemo << 0b1100 2            → 0b110000
bitdemo >> 0b1100 2            → 0b11
>> -8 2                        → -2
```

❶ See `format` (Section 4.20) and argument expansion (Section 3.4)

7.2.2. Infix expressions: expr

```
expr ARG ?ARG …?
```

The commands discussed in the previous section provide one means for numeric computation in Tcl. However, unless you are from the Lisp world, you might find it awkard to compute `2+3*4` as

```
set val [+ 2 [* 3 4]] → 14
```

The Tcl expression syntax allows the more common infix syntax

```
2+3*4
```

as an alternative. However, because of Tcl's uniform syntax where interpretation of arguments is entirely up to the command itself, **in-fix notation can only be used with commands that interpret arguments as expressions**.

This can be a point of confusion so, at the risk of belaboring the point, let us take a couple of examples.

```
set val 2+3*4   → 2+3*4
list $val == 14 → 2+3*4 == 14
```

The first statement above assigns the **string** `2+3*4` to the variable `val`, not the result of that expression. Similarly, the `list` command in the second statement creates a list from its arguments `2+3*4`, `==` and `14`.

In contrast, commands like `expr` will treat the arguments as in-fix expressions.

```
expr 2+3*4 → 14
```

Here the `expr` command interprets its argument as an expression and returns the result.

Thus unlike most other languages, Tcl will not necessarily treat a string of characters that looks like a numeric expression as being one. The interpretation is dependent on the command that receives it as an argument.

We will describe expressions within the context of `expr` since it is the most fundamental of these commands but keep in mind the discussion applies to all commands that accept expressions as arguments. These include `if` (Section 3.7), `while` (Section 3.9) and `for` (Section 3.10).

The command concatenates the supplied arguments separating them by space characters. The result is then treated as a *Tcl expression* and evaluated. The syntax of a Tcl expression differs from the normal Tcl syntax and is described next.

Expressions consists of

- comments
- operands which are the values to be used in the operations
- operators which define the operations to be executed
- parenthesis for grouping operands

We describe each of these in turn.

7.2.2.1. Comments in expressions

Expressions may contain embedded comments. A `#` character at any point within an expression, except within double quotes or braces, starts a comment. The comment includes all characters until the end of the line or end of the expression, whichever comes first.

```
set radius 3
expr {2         # Circumference is 2
      *         # times
      3.142     # pi
      * $radius # times radius}
→ 18.852
```

Note the terminating brace above is not treated as part of the comment.

7.2.2.2. Operands in expressions

An operand in an expression may be one of

- a number in any of the representations described earlier.
- a boolean literal in any form such as `true`, `false` etc.
- a Tcl variable, dereferenced as usual with a `$` prefix.
- a "double quoted" string (Section 3.3.1). The expression evaluator will do the same backslash, variable, and command substitutions (Section 3.2) on the string as Tcl.
- a brace quoted string (Section 3.3.2) which is again parsed as in Tcl.
- a Tcl command enclosed in brackets. The result of the command is used as the operand value.
- a mathematical function that uses the form `func(arg,…)` where the arguments have the same syntax as other operands but are separated by commas.

Some examples:

```
expr {[clock seconds] + 86400} → 1743780920 ❶
set exponent 3                 → 3
expr {pow(2,$exponent)}        → 8.0 ❷
set s "bar"                    → bar
expr {"foobar" eq "foo$s"}     → 1 ❸
expr {"foobar" eq {foo$s}}     → 0 ❹
```

❶ Bracketed command
❷ Call to the math function `pow` with numeric literal and variable arguments
❸ Double quoted literal string operand
❹ Brace quoted literal string operand

One point to make a special note of is that unlike Tcl, **the expression mini-language requires string literals to be quoted within expressions**. For example,

```
% expr {bar eq $s} ❶
Ø invalid bareword "bar"
  in expression "bar eq $s";
  should be "$bar" or "{bar}" or "bar(...)" or ...
% expr {"bar" eq $s} ❷
→ 1
```

❶ Error - string literals must be enclosed in quotes or braces
❷ Ok - `bar` is placed in quotes

7.2.2.3. Operators in expressions

The operators supported in expressions are shown in Table 7.7. They include those we described previously in Section 7.2.1 and a few more.

The table lists the operators them in order of descending *precedence*. Operators with higher precedence are evaluated before those with lower precedence. For example, in the expression `2+3*4`, the `3*4` is evaluated first as `*` has a higher precedence than `+` as shown in the table. Operators at the same precedence level are evaluated left to right **excepting the exponentiation operator as noted**.

Table 7.7. Expression operators in precedence order

Operators	Description
`-`, `+`, `~`, `!`	Unary operators. The `-` and `+` indicate the sign of a numeric operand. The `~` is a bit-wise complement and may only be applied to integer operands. The `!` operator is a logical complement and may be applied to both boolean and numeric operands.
`**`	Exponentiation (Section 7.2.1.1).
`*`, `/`, `%`	Multiplication, division and remainder operators. See Table 7.2 regarding the sign of the remainder for the `%` operator.
`+`, `-`	Addition and subtraction.
`<<`, `>>`	Left and right shifts. Only valid for integer operands. Right shifts propagate the sign bit.
`<`, `<=`, `>`, `>=`	Polymorphic comparison operators (Section 7.2.1.2).
`lt`, `le`, `gt`, `ge`	String comparison operators (Section 7.2.1.3).
`==`, `!=`	Polymorphic equality operators (Section 7.2.1.2).
`eq`, `ne`	String equality operators (Table 7.4).
`in`, `ni`	List containment operators (Section 7.2.1.4).
`&`	Bit-wise AND operation (Section 7.2.1.5).
`^`	Bit-wise XOR operation (Section 7.2.1.5).
`|`	Bit-wise OR operation. (Section 7.2.1.5).
`&&`	Logical AND operation on booleans or numbers (interpreted as booleans). The evaluation is "short-circuited" (see below).
`||`	Logical OR operation on booleans or numbers (interpreted as booleans). The evaluation is "short-circuited".

`?:`	This is the conditional operator that takes the form
	`CONDITION ? TRUEVAL : FALSEVAL`
	If the *CONDITION* operand evaluates to true, the result of the expression is `TRUEVAL`; otherwise, it is `FALSEVAL`.
	`% proc min {a b} { expr {$a <= $b ? $a : $b} }`
	`% min 2 -2`
	`→ -2`

The operators `&&`, `||` and `?:` undergo "short-circuited" evaluation in that some arguments may not be evaluated if the value of the expression is already determined. In the case of the `&&` operator, if the first operand evaluates to `false`, the second operand is not evaluated.

```
expr {2 > 3 && [nosuchcommand]} → 0
```

The `>` has a higher precedence than && is therefore executed first. Since it evaluates to a boolean false, the second operand of the && is never evaluated and consequently no error is raised about the command not existing.

Similarly, in the case of the `||` operator, if the first argument evaluates to `true`, the second argument is not evaluated. In the case of the `?:` operator, if *`CONDITION`* evaluates to `false`, then the *`TRUEVAL`* operand is never evaluated; likewise for *`FALSEVAL`* if *`CONDITION`* is `true`.

7.2.2.4. Grouping operands with parenthesis

To change the order of evaluation in an expression from the default operator precedence order, use parenthesis to force a different order.

```
expr 2+3*4   → 14
expr (2+3)*4 → 20
```

7.2.2.5. Braces and double substitution

It is important to note that the arguments passed to `expr` are parsed twice, once by the Tcl parser and once by the command itself. This may lead to double substitution. Consider the following code.

```
set elem a
set lst {a b c d}
expr {$elem in $lst}
→ 1
```

Here because the braces protect against substitutions by Tcl, the `expr` command sees a single argument `$elem in $list`. It parses this argument as per its rules, doing variable substitution etc. and returns the result.

On the other hand, the following raises an error exception.

```
expr $elem in $lst
Ø invalid bareword "a"
  in expression "a in a b c d";
  should be "$a" or "{a}" or "a(...)" or ...
```

Now because there are no braces to protect substitutions, the Tcl parser will substitute the variables. The `expr` command sees three arguments, `a`, `in` and `a b c d` and as per its defined behaviour concatenates them to form the expression `a in a b c d` which is not a valid expression.

This example is illustrative of the fact that you have to be aware of the potential for double substitution. In rare cases this may be desirable, but in most instances **you are strongly advised to use the braced argument form of the `expr` invocation** for reasons of performance and safety:

- Braced expressions can be compiled and cached internally for significantly better performance. So for example, using the `timerate` command (Section 26.2.3) for measuring execution time, we can see

```
% timerate -calibrate {}
→ 0.03771306637973359 µs/#-overhead 0.038824 µs/# 50278809 # 25757586 #/sec
% set x 1 ; set y 2 ; set z 3
→ 3
% timerate {expr $x+$y*$z}
→ 1.557279 µs/# 626968 # 642145 #/sec 976.364 net-ms
% timerate {expr {$x+$y*$z}}
→ 0.241218 µs/# 3585113 # 4145621 #/sec 864.795 net-ms
```

- Double substitution opens your code to unexpected surprises and even security risks, similar to SQL injection attacks, if the expression being evaluated comes from some untrusted source. Consider writing a program that will print the double of a number input by the user which is stored in the variable `input`. Either forms of the `expr` command below returns the right result.

```
set input 3     → 3
expr $input*2   → 6
expr {$input*2} → 6
```

  Now however imagine the user inputs the following string instead.

```
% set input {[puts Hacked!; string cat 3]}
→ [puts Hacked!; string cat 3]
% expr $input*2
→ Hacked!
  6
% expr {$input*2}
Ø cannot use a list as left operand of "*"
```

  Now you can see the problem with the unbraced version which lands up executing the `puts` command. Imagine if that command was something more nefarious that formatted your disk via `exec`. The braced form of the command does not suffer from this vulnerability, generating an error instead.

Generally speaking, you should limit yourself to using the unbraced argument form only in interactive mode where it is a little more convenient.

The safest way to evaluate expressions from untrusted sources is through **safe interpreters** which we will study in Section 23.10.

7.2.3. Incrementing variables: incr

```
incr VAR ?INCREMENT?
```

The `incr` command targets the common case of incrementing variables holding integers. Here `VAR` is the name of a variable that, if already existing, must hold an integer value. If it does not exist, it is created with an initial value of `0`. The value is then incremented by `INCREMENT` which must also be an integer and defaults to `1`. The command returns the new value of `VAR`.

```
incr newvar -2 → -2 ❶
incr newvar    → -1 ❷
```

❶ `newvar` created with value of 0 and then decremented
❷ Default increment of `1`

The command is roughly equivalent to

```
set VAR [expr {$VAR + INCREMENT}]
```

except that it only works with integers and not floating point numbers.

More than just conciseness, the advantages of `incr` compared to `expr` is efficiency as `incr` modifies the variable "in place" as opposed to generating a new value that is assigned back to the variable.

You will sometimes find the following apparent no-op in Tcl code:.

```
incr ival 0
```

The purpose of this statement is to verify that `ival` holds an integer value. If not, an error will be raised. This is both faster and less verbose than

```
if {[string is entier -strict $ival]} {
    error "Expected integer, got \"$ival\""
}
```

7.2.3.1. Expressions in other commands

The `expr` command is not the only command that uses expression syntax. Several other commands such as `if` (Section 3.7), `while` (Section 3.9) and `for` (Section 3.10) also use expression syntax for their condition argument. For example,

```
if {$n > 0} {…some code…}
```

As described in Section 3.9, enclosing the condition in braces has added importance there.

7.3. Mathematical functions

Tcl provides some commonly used mathematical functions shown in Table 7.8 as commands in the `::tcl::mathfunc` namespace. We will not detail these commands here as their functionality should be obvious. See the Tcl reference documentation of `mathfunc` for details.

Table 7.8. Mathematical functions

Functions	Description
`bool`, `int`, `wide`, `entier`, `double`	Number conversion. See Section 7.1.5.2.
`abs`, `ceil`, `floor`, `fmod`, `round`, `max`, `min`	Miscellaneous numeric functions.
`exp`, `pow`, `log`, `log10`, `isqrt`, `sqrt`	Functions related to exponents and logarithms.
`sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `atan2`, `hypot`	Trigonometric and geometric functions.
`rand`, `srand`	Functions for random number generation.
`isfinite`, `isinf`, `isnan`, `isnormal`, `issubnormal`, `isunordered`	Functions for classifying floating point values. Also see Section 7.1.3.1. **NOTE:** These functions are not available with Tcl 8.6.

You can also enumerate the available functions with the `info functions` command.

```
% info functions
→ round wide isinf sqrt sin log10 isfinite double hypot atan bool rand isnormal...
% info functions log*
→ log10 log
```

These commands can be called like any other command by qualifying with the `tcl::mathfunc` namespace. Alternatively you can import (Section 16.5.3.1) the commands or add the `::tcl::mathfunc` namespace to the namspace path (Section 16.5.3.2) to call the commands without qualification.

```
tcl::mathfunc::rand      → 0.6293399928274285
tcl::mathfunc::round 3.5 → 4
```

7.3.1. Using functions in expressions

Commands in the `::tcl::mathfunc` namespace can be called from expressions using the special function call expression syntax described in Section 7.2.2.2. Note that when called in this manner, the function name does not need to be qualified even if it is not imported or present on the namespace path.

```
set pi 3.14159    → 3.14159
expr {sin($pi/2)} → 0.9999999999991198
```

7.3.2. Defining custom functions

You can add your own commands to the `::tcl::mathfunc` namespace. Doing so allows you to use the command as a function in an expression.

```
proc ::tcl::mathfunc::signum n {expr {$n < 0 ? -1 : $n > 0 ? 1 : 0}}
expr {signum(-5)}
→ -1
```

Naturally, you have to take care that the functions you add do not clash with those that might be added by other libraries in the application or even by Tcl itself in the future. It is best to prefix the name appropriately to reduce the chance of a collision.

8

Binary data

> When you're young, you think everything has to be binary...
>
> — Min Jin Lee

Many applications, such as those implementing network protocols, data compression etc. need to work with binary data where the individual bytes, or even bits, are manipulated without any notion of "characters". Such data is handled in Tcl as *binary strings* through specific commands targeted towards their manipulation. As a script level abstraction, these binary strings are nothing but ordinary Tcl strings (Section 4.1) but with the Unicode code points restricted to the range U+0000-U+00FF (thus each fitting in a byte). Most `string` commands such as `string length`, `string index` etc. work naturally with binary strings. For constructing and parsing binary strings, and conversion to common human readable encodings such as base64, Tcl provides the `binary` ensemble command. Since compression of binary data is a common operation, Tcl also provides the `zlib` command for the purpose.

For ease of displaying binary data, we will first define a simple (but not the most efficient) procedure, bin2hex, using the `binary encode` command that we will see later.

```
proc bin2hex {args} {
    puts [regexp -inline -all .{2} [binary encode hex [join $args ""]]]
}
```

This will print each byte in a binary string in hexadecimal format.

The above illustrates a quick and dirty method of using `regexp` (Section 10.1) for splitting strings into equal size chunks.

8.1. Binary literals

The easiest way to create simple binary string constants with known content is to use the `\x` syntax. In the example below, `bin` is set to a string that is a sequence of the three Unicode code points U+0001, U+0080 and U+00FF. Conceptually, when working with the string as binary data, it can also be viewed simply as a sequence of three bytes `01`, `80`, `ff` (in hex).

```
set bin "\x01\x80\xff"
bin2hex $bin
→ 01 80 ff
```

The binary string can be operated on with the usual string manipulation commands.

```
string length $bin → 3
```

Creating a binary literal is even easier if it contains only 7-bit values corresponding to ASCII characters.

```
bin2hex "XYZ" → 58 59 5a
```

8.2. Encoding binary strings as ASCII

There are times when binary data has to be encoded into 7- or 8-bit ASCII form for transport through email, for human readability, storage in files based on ASCII encodings and so on.

There are three commonly used ASCII based formats used encoding binary data — plain hexadecimal encoding, base64 and uuencode. All three are supported by the `binary encode` and `binary decode` commands.

8.2.1. Hexadecimal format: binary encode | decode hex

```
binary encode hex BINDATA
binary decode hex ?-strict? ENCODED
```

The `binary encode hex` and `binary decode hex` commands convert binary strings to and from hexadecimal ASCII strings. Each byte of *`BINDATA`* is encoded as a pair of hex digits, most significant nibble first.

```
binary encode hex XYZ            → 58595a
binary encode hex "\xfe\xf0\x0f" → fef00f
binary decode hex "58595a"       → XYZ
```

An error is raised if the argument contains anything other than hexadecimal characters and whitespace. If passed the `-strict` option, even whitespace is not permitted.

```
% binary decode hex "58 595a"
→ XYZ
% binary decode hex -strict "58 595a"
Ø invalid hexadecimal digit " " (U+000020) at position 2
```

8.2.2. Base64 format: binary encode | decode base64

```
binary encode base64 ?-maxlen MAXLEN? ?-wrapchar CHAR? BINDATA
binary decode base64 ?-strict? ENCODED
```

The `binary encode base64` and `binary decode base64` commands convert binary strings to and from base64 encoded ASCII strings.

If the `-maxlen` option is specified, strings longer than *`MAXLEN`* are split into multiple lines. The `-wrapchar` option may be specified to split the string using another character in lieu of a newline. As for hexadecimal decoding, whitespace is ignored unless the `-strict` option is present.

```
% binary encode base64 "\xfe\xf0\x0f"
→ /vAP
% set enc [binary encode base64 -maxlen 30 [string repeat XYZ 20]]
→ WFlaWFlaWFlaWFlaWFlaWFlaWF
  laWFlaWFlaWFlaWFlaWFlaWFla
  WFlaWFlaWFlaWFlaWFla
% binary decode base64 $enc
→ XYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZ
```

Passing `-strict` when decoding will however generate an error since the encoded output above was wrapped with newlines.

```
% binary decode base64 -strict $enc
Ø invalid base64 character "
  " (U+00000A) at position 30
```

8.2.3. Uuencode format: binary encode|decode uuencode

```
binary encode uuencode ?-maxlen MAXLEN? ?-wrapchar CHAR? BINDATA
binary decode uuencode ?-strict? ENCODED
```

The final form of binary to ASCII encoding, the *uuencode* format, is implemented by the `binary encode uuencode` and `binary decode uuencode` commands.

This supports the `-maxlen` and `-wrapchar` options described for the base64 encoding.

```
% binary encode uuencode "\xfe\xf0\x0f"
→ #_O`/
% set enc [binary encode uuencode -maxlen 30 [string repeat XYZ 20]]
→ 56%E:6%E:6%E:6%E:6%E:6%E:6%E:
  56%E:6%E:6%E:6%E:6%E:6%E:6%E:
  26%E:6%E:6%E:6%E:6%E:6%E:
% binary decode uuencode $enc
→ XYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZ
```

Note however, that the `-strict` option for uuencode differs in that it only throws an error if whitespace appears in unexpected places as the format itself allows for it in some locations.

8.3. Constructing binary strings: binary format

```
binary format FORMATSTRING ?ARG ARG …?
```

The `binary format` command is used to construct a binary string in a similar fashion to how the `format` (Section 4.20) command is used for constructing character strings.

The *FORMATSTRING* argument specifies the structure or layout of the binary string as a sequence of fields of various types and sizes. The returned binary string is constructed by filling each field with the value of the corresponding argument formatted appropriately.

As an introductory example, consider the initial part of a TCP header for an network connection, which consists of

- a 16-bit source port number, say 5000 or 0x1388 in hex
- a 16-bit destination port number 80, or 0x0050 hex
- a 32-bit sequence number, say 1000000, or 0x000F4240 hex
- a 32-bit acknowledgement number, say 100 or 0x00000064 hex

All fields are sent in network byte order (big endian, most significant byte first) so the stream of bytes would appear in hexadecimal as

```
13 88 00 50 00 0F 42 40 00 00 00 64
```

Within a Tcl script the above header fields might be stored in variables and the `binary format` command used to construct the packet header.

```
set srcport 5000
set dstport 80
set seqnum 1000000
set acknum 100
set header [binary format SSII $srcport $dstport $seqnum $acknum]
bin2hex $header
→ 13 88 00 50 00 0f 42 40 00 00 00 64
```

The format types `S` and `I` specify 16-bit big endian and 32-bit big endian fields as per the desired layout. Because the constructed header contains non-printable data, we use our `bin2hex` wrapper around the `binary encode hex` command to display it in printable hexadecimal form.

The format string may include spaces for readability purposes. In the above example the format string `SSII` may have been specified as `S S I I` or `SS I I` etc. with no difference in the generated binary string.

In general, *FORMATSTRING* should be a sequence of field specifiers each of which is

- a single character that either specifies a type or a cursor movement
- optionally followed by an flag character
- optionally followed by a numeric count field

The type and cursor specifiers are detailed later in Section 8.3.1 and Section 8.3.2.

The flag character, for which `u` is the only valid value, is ignored and not discussed here. It is accepted by the `binary format` command only for compatibility with the `binary scan` command allowing the same format string to be used for both.

The count field may be either a positive integer value or the character `*`. An integer value specifies the number of fields of that type to be placed at that position. The values are picked up from the corresponding argument which may be a string or a list depending on the type

specifier. The `*` character works similarly except that it indicates that all the values in the corresponding argument are to be used.

Thus our previous `binary format` example could also have been written as follows.

```
bin2hex [binary format "S2 I*" [list $srcport $dstport] [list $seqnum $acknum]]
→ 13 88 00 50 00 0f 42 40 00 00 00 64
```

8.3.1. Type specifiers for binary format

The type characters, such a `S` or `I` in our example and summarized in Table 8.1, indicate both the type (integer, real etc.) of a field as well as its layout (width, endianness).

Table 8.1. Type specifiers for binary format

Specifiers	Description
`a`, `A`	Byte string padded with null bytes and binary value 32/0x20 (ASCII space) respectively.
`b`, `B`	Bit string. Arguments must be a string of binary digits `0` and `1`. Packed within each output byte in low to high and high to low order respectively.
`h`, `H`	String of hexadecimal digits packed in each byte in low to high and high to low order respectively.
`c`	List of integers if a count is specified. Only the low order 8 bits are stored in the output byte.
`s`, `S`, `t`	List of integers. Only the low order 16 bits are stored in the output in little endian, big endian and native order respectively.
`i`, `I`, `n`	List of integers. Only the low order 32 bits are stored in the output in little endian, big endian and native order respectively.
`w`, `W`, `m`	List of integers. Only the low order 64 bits are stored in the output in little endian, big endian and native order respectively.
`r`, `R`, `f`	List of single precision floating point numbers. Stored in little endian, big endian and native order respectively.
`q`, `Q`, `d`	List of double precision floating point numbers. Stored in little endian, big endian and native order respectively.
`x`	Stores zeroes in the output.

Binary formats: `a`, `A`

The character `a` specifies a single byte field. The argument is a character string and the value stored in the field is taken from the **low 8 bits** of the Unicode code point for the corresponding character with higher bits discarded.

```
bin2hex [binary format a z]      → 7a
bin2hex [binary format a \u0102] → 02 ❶
```

❶ Note truncation to low 8 bits.

If a count specifier is present, the appropriate number of characters from the argument string are used. Any extra characters in the argument are ignored. If the argument has fewer characters than the specified count, the remaining bytes are filled with null bytes.

```
bin2hex [binary format a3 wxyz]  → 77 78 79 ❶
bin2hex [binary format a3a yz x] → 79 7a 00 78 ❷
```

❶ Only 3 characters used.
❷ Note padding first argument with nulls.

The specifier `A` is similar to `a` except that if the string argument has fewer characters than the specified count, the remaining bytes are filled with the binary value 32/0x20 (an ASCII space) instead of null bytes.

```
bin2hex [binary format A*A3 wxyz yz] → 77 78 79 7a 79 7a 20 ❶
```

❶ Note padding with spaces.

Binary format: `b`, `B`

Arguments must be a string of binary digits `0` and `1`. For `b`, these are packed into output bytes in low to high order within each byte. Zeroes are used if the argument string is shorter than the count for a field or if the number of bits is not a multiple of 8. `B` is similar except that bits are stored in high to low order within a byte.

```
bin2hex [binary format b8 10101010]         → 55
bin2hex [binary format B8 10101010]         → aa ❶
bin2hex [binary format "b8 b5" 101 11111]   → 05 1f ❷
bin2hex [binary format "B8 B5" 101 11111]   → a0 f8 ❸
bin2hex [binary format b* 1011001110001110] → cd 71 ❹
```

❶ Note different output bit order from above.
❷ Zero fill high bits.
❸ Zero fill low bits.
❹ Output as many bytes as needed.

Binary format: `h`, `H`

The argument is a string of hexadecimal digits. Both lower and upper case characters are accepted. In the case of `h`, the hex digits are packed in the output bytes in low to high order whereas for `H` they are packed in the high to low order which is what is normally desired. Zeroes are used to fill if the argument string is shorter than the count for a field or if the number of hexadecimal characters is odd.

```
bin2hex [binary format h* 0aB] → a0 0b
bin2hex [binary format H* 0aB] → 0a b0
```

Binary format: `c`

If a count is not specified, the argument must be an integer the low 8 bits of which are stored in the byte. If count is specified, the argument must be a list of *at least* that many integers. The generated output is then a sequence of bytes each containing the low 8 bits of the corresponding integer element.

```
bin2hex [binary format cc2 10 {-1 1}]        → 0a ff 01
bin2hex [binary format c* {254 255 256 257}] → fe ff 00 01 ❶
```

❶ Note truncation to low 8 bits.

Binary format: `s`, `S`, `t`

If a count is not specified, the argument must be an integer the low 16 bits of which are stored in two bytes in little endian, big endian and native order for `s`, `S` and `t` respectively. If count is specified, the argument must be a list of *at least* that many integers. The generated output is then a sequence of bytes each containing the low 16 bits of the corresponding integer element.

```
bin2hex [binary format ss* 33825 {-2 65537}] → 21 84 fe ff 01 00
bin2hex [binary format SS* 33825 {-2 65537}] → 84 21 ff fe 00 01
bin2hex [binary format tt* 33825 {-2 65537}] → 21 84 fe ff 01 00
```

Binary format: `i`, `I`, `n`

Similar to `s` except that `i`, `I` and `n` store 32-bit integers in 4 byte output sequences in little endian, big endian and native order respectively.

```
bin2hex [binary format ii* 2151678465 {-2 65537}] → 01 02 40 80 fe ff ff ff 01 00
 01 00
bin2hex [binary format II* 2151678465 {-2 65537}] → 80 40 02 01 ff ff ff fe 00 01
 00 01
```

Binary format: `w`, `W`, `m`

Similar to `s` except that `w`, `W` and `m` store 64-bit integers in 8 byte output sequences in little endian, big endian and native order respectively.

```
bin2hex [binary format w 18049651735527937] → 01 02 04 08 10 20 40 00
bin2hex [binary format W 18049651735527937] → 00 40 20 10 08 04 02 01
```

Binary format: `r`, `R`, `f`

Store single precision floating point numbers in little endian, big endian and native order respectively. The number of bytes produced depends on machine architecture.

```
bin2hex [binary format r 2.71828] → 4d f8 2d 40
bin2hex [binary format R 2.71828] → 40 2d f8 4d
```

Binary format: `q`, `Q`, `d`

Store double precision floating point numbers in little endian, big endian and native order respectively. The number of bytes produced depends on machine architecture.

```
bin2hex [binary format q 2.71828] → 90 f7 aa 95 09 bf 05 40
bin2hex [binary format Q 2.71828] → 40 05 bf 09 95 aa f7 90
```

Binary format: `x`

Stores zeroes in the output. This differs from the other types in that it does not consume an argument and does not permit the count to be specified as `*`.

```
bin2hex [binary format cxcx2c 255 254 253] → ff 00 fe 00 00 fd
```

8.3.2. Cursor movement for formatting

The `binary format` command writes output bytes at a position in the string indicated by a *cursor*. In addition to the type specifiers just discussed, the format specification can include cursor movement characters. Normally the cursor is positioned at the end of the last output sequence. Cursor movement characters change the position of this cursor and unlike type specifiers do not consume any arguments.

Table 8.2. Binary format cursor movement characters

Specifier	Description

`X`	Moves the cursor backward by the specified count or by one character if count is not specified. If the count is `*` or greater than the current position, the cursor is moved to the beginning.
	`bin2hex [binary format c3c2 {0 1 2} {3 4}]  → 00 01 02 03 04`
	`bin2hex [binary format c3X2c2 {0 1 2} {3 4}] → 00 03 04`

`@`	Moves the cursor to the absolute position given by count which must be specified. If the count is greater than the current output length, the output is padded with zero bytes. If the count is `*`, the cursor is placed at the end of the string. `bin2hex [binary format c5@2c2@*c {0 1 2 3 4} {5 6} 7] → 00 01 05 06 04 07`

8.4. Parsing binary strings: binary scan

```
binary scan BINSTRING SCANFORMAT ?VARNAME VARNAME …?
```

The `binary scan` command is used to parse a binary string in a similar fashion to how the `scan` (Section 4.21) command is used for parsing character strings. It is conceptually the inverse of the `binary format` (Section 8.3) command.

The parse of binary string *BINSTRING* is driven by a format string *SCANFORMAT* that specifies the expected structure or layout of *BINSTRING* as a sequence of fields of various types and sizes. The values are extracted and stored in the *`VARNAME`* variables. The command returns the number of variables that were set.

The following code parses the binary TCP header from the previous section.

```
% bin2hex $header
→ 13 88 00 50 00 0f 42 40 00 00 00 64
% binary scan $header SSII scan_srcport scan_dstport scan_seq scan_ack
→ 4
% puts "$scan_srcport, $scan_dstport, $scan_seq, $scan_ack"
→ 5000, 80, 1000000, 100
```

The syntax of the *SCANFORMAT* argument is the same as the format specifiers used for the `binary format` command. It is a sequence of field specifiers each of which is

- a single character that either specifies a type or a cursor movement,
- optionally followed by an flag character,
- optionally followed by a numeric count field.

The field specifiers may be separated by spaces.

The scan begins at the start of the input binary string and maintains a cursor position within the string that is updated after each field specifier. If the field specifier denotes a type, the bytes following the cursor position are scanned as binary data of that type and the cursor is moved to point to the following byte. If the field specifier denotes cursor movement, the cursor is moved without any bytes being scanned.

The flag character, for which `u` is the only valid value, may be specified with any type but only has effect for certain integer types where it marks the field to be interpreted as an unsigned value. For example,

```
% bin2hex [set bin [binary format i 0xffffffff]]
→ ff ff ff ff
% binary scan $bin i value; puts $value ❶
→ -1
% binary scan $bin iu value; puts $value ❷
→ 4294967295
```

❶ `i` specifies 32-bit little endian integer
❷ `iu` specified *unsigned* 32-bit little endian integer

The count field may be

- a positive integer value in which case it specifies the number of fields of that type to be parsed and stored in the corresponding variable
- the character `*` which indicates that all the remaining bytes are to be parsed as that type

The binary string being parsed may not have sufficient bytes to satisfy the scan string specification. This is not treated as an error. Instead as many field specifiers as can be *fully* satisfied are parsed and stored in the corresponding variables. Remaining variables are not affected.

```
% binary scan $header SSIII scan_srcport scan_dstport scan_seq scan_ack extra_var
→ 4
% puts "$scan_srcport, $scan_dstport, $scan_seq, $scan_ack"
→ 5000, 80, 1000000, 100
% puts [info exists extra_var]
→ 0
```

8.4.1. Type specifiers for binary scan

The type character, such a `S` or `I` in our example, indicates both the type (integer, real etc.) of a field as well as its layout (width, endianness). Table 8.3 shows the various type specifiers available.

Table 8.3. Type specifiers for binary scan

Specifier	Description
`a`, `A`	Extract a single byte differing in their treatment of trailing spaces and zero bytes. The byte is treated as a Unicode character in the range U+0000-U+00FF.
`C`	Extracts all bytes until a zero byte is encountered as a string mapping each byte to a Unicode character in the range U+0000-U+00FF. **NOTE:** Not available in Tcl 8.6 and earlier.
`b`, `B`	Extract bits in a byte in low to high and high to low order respectively.
`h`, `H`	Extract the nibbles of a byte as a pair of hexadecimal digits in low to high or high to low order respectively.
`c`	Extracts bytes as signed 8-bit integers or unsigned if the `u` flag is specified.
`s`, `S`, `t`	Extract pairs of bytes as 16-bit signed, or unsigned if the `u` flag is specified, integers in little endian, big endian and native order respectively.
`i`, `I`, `n`	Extract pairs of bytes as 32-bit signed, or unsigned if the `u` flag is specified, integers in little endian, big endian and native order respectively.
`w`, `W`, `m`	Extract pairs of bytes as 64-bit signed, or unsigned if the `u` flag is specified, integers in little endian, big endian and native order respectively.
`r`, `R`, `f`	Extract single precision floating point numbers stored in little endian, big endian and native order respectively.
`q`, `Q`, `d`	Extract double precision floating point numbers stored in little endian, big endian and native order respectively.

Details and examples of each format are below.

Binary scan: `a`, `A`

The `a` specifier denotes a single byte field. The value is stored as a Unicode character in the range U+0000-U+00FF. The `A` specifier is similar with the solitary difference that **trailing** spaces and zero bytes are stripped from **each** value stored.

```
set bin "abc   def  "           → abc   def ❶
binary scan $bin a5a* val1 val2 → 2
puts "<$val1>, <$val2>"         → <abc  >, < def  >
binary scan $bin A5A* val1 val2 → 2
puts "<$val1>, <$val2>"         → <abc>, < def>
```

❶ For pure ASCII this is the same as `[binary format a* "abc def "]`

Binary scan: `C`

This specifier is primarily intended for extracting C style nul-terminated strings embedded within binary data. It extracts all bytes up to the next zero byte, or end of data, and stores it as a string formed by mapping each byte value to the same numeric Unicode code point. The cursor is positioned after the zero byte which is not included in the stored string.

```
binary scan "abc\0def" CC val1 val2 → 2
puts "<$val1>, <$val2>"             → <a>, <b>
```

Binary scan: `b`, `B`

The `b` specifier parses bits in a byte in low to high order storing them in the variable as a string of `0` and `1` characters. The `B` specifier is similar except that the bits are processed in high to low order within a byte.

```
binary scan "\x00\x5f\xaa" b13b* val1 val2 → 2
puts "$val1, $val2"                        → 0000000011111, 01010101
binary scan "\x00\x5f\xaa" B13B* val1 val2 → 2
puts "$val1, $val2"                        → 0000000001011, 10101010
```

Note how each field specifier always begins at a byte boundary. The first specifier maps 13 bits. The remaining 3 bits to the next byte are skipped since the next specifier will only start at the next byte boundary.

Binary scan: `h`, `H`

Parses the binary data into a string of hexadecimal digits. The digits are taken from low to high order for each byte for `h` and the (natural) high to low order for `H`.

```
binary scan "\xab\xcd\xef" H3H* val1 val2 → 2
puts "$val1, $val2"                       → abc, ef
binary scan "\xab\xcd\xef" h3h* val1 val2 → 2
puts "$val1, $val2"                       → bad, fe
```

Again, note how each field specifier always begins at a byte boundary.

Binary scan: `c`

The byte(s) in the binary string are converted to signed 8-bit integers and stored in the corresponding variable as a list. Adding the `u` flag stores treates the bytes as unsigned 8-bit integers.

```
binary scan \xff\x00\x01\xfe\x0f\x80 cc2c* var1 var2 var3   → 3
puts "$var1, $var2, $var3"                                 → -1, 0 1, -2 15 -128
binary scan \xff\x00\x01\xfe\x0f\x80 cuc2cu* var1 var2 var3 → 3
puts "$var1, $var2, $var3"                                 → 255, 0 1, 254 15 128
```

Binary scan: `s`, `S`, `t`

The data is interpreted as 16-bit signed integers stored in little endian, big endian and native order respectively. As for the `c` specifier, adding the `u` flag results in a field being treated as unsigned.

```
binary scan \xff\x00\x00\xff\xff\x00\x00\xff s2su* val1 val2 → 2
puts "$val1, $val2"                                         → 255 -256, 255 65280
binary scan \xff\x00\x00\xff\xff\x00\x00\xff S2Su* val1 val2 → 2
puts "$val1, $val2"                                         → -256 255, 65280 255
```

Binary scan: `i`, `I`, `n`

The data is interpreted as 32-bit signed integers stored in little endian, big endian and native order respectively. Adding the `u` flag results in a field being treated as unsigned.

```
binary scan \x00\x00\x00\xff\x00\x00\x00\xff iiu val1 val2 → 2
puts "$val1, $val2"                                       → -16777216, 4278190080
```

Binary scan: `w`, `W`, `m`

The data is interpreted as 64-bit signed integers stored in little endian, big endian and native order respectively. Adding the `u` flag results in a field being treated as unsigned.

```
binary scan \xff\x00\x00\x00\x00\x00\x00\x00 wu val1 → 1
puts "$val1"                                        → 255
binary scan \xff\x00\x00\x00\x00\x00\x00\x00 W val1  → 1
puts "$val1"                                        → -72057594037927936
```

Binary scan: `r`, `R`, `f`

The data is interpreted as single precision floating point numbers stored in little endian, big endian and native order respectively.

```
bin2hex [set bin [binary format r 2.71828]] → 4d f8 2d 40
binary scan $bin r e                        → 1
puts "$e"                                   → 2.718280076980591
```

The difference of course stems from floating point representation rounding errors.

Binary scan: `q`, `Q`, `d`

The data is interpreted as double precision floating point numbers stored in little endian, big endian and native order respectively.

```
bin2hex [set bin [binary format q 3.14159]] → 6e 86 1b f0 f9 21 09 40
binary scan $bin q pi                       → 1
puts "$pi"                                  → 3.14159
```

8.4.2. Cursor movement for scanning

In addition to the field types, the scan specification can include the cursor movement characters shown in Table 8.4 that control the scan position for the next specifier.

Table 8.4. Binary scan cursor movement characters

Specifiers	Description
`x`	Moves the cursor forward.
`X`	Moves the cursor backward.
`@`	Moves the cursor to an absolute position.

Binary scan: `x`

Moves the cursor forward by one byte or the specified count. If count is specified as `*` or is larger than the remaining byte count, the cursor is placed at the end of the input binary string.

```
binary scan \x01\x02\x03\x04\x05\06 cxcx2c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                               → 1, 3, 6
```

Binary scan: `X`

Moves the cursor backward by the specified count or by one if no count is specified. If the count is `*` or greater than the current position, the cursor is placed at the start.

```
binary scan \x01\x02\x03\x04\x05\06 c2Xc3X2c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                                  → 1 2, 2 3 4, 3
```

Binary scan: `@`

Moves the cursor to the absolute position given by count which must be specified. If the count is greater than the current output string length, the cursor is placed at the end of the string.

```
binary scan \x01\x02\x03\x04\x05\06 c2@0c3@5c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                                   → 1 2, 1 2 3, 6
```

8.5. Compressing data

A common operation on binary data is compression for purposes of saving on storage space or speed of transmission; `zip` and `gz` archives are examples of the former and HTTP compression an example of the latter. Tcl provides commands implementing compression and decompression based on the `zlib` library of data compression algorithms and the *zip* and *gzip* file compression formats that are used in most applications.

The zlib family really consists of three different specifications:

- The raw compression algorithm, DEFLATE, defined in RFC 1951[1]. We refer to data compressed using this algorithm as *deflated* data.
- A data format, the *ZLIB compressed data format*, defined in RFC 1950[2] that wraps the raw deflated data to include additional metadata such as checksums. We refer to this as *zlib compressed* data.
- Another data format, the *GZIP file format*, defined in RFC 1952[3] that also wraps the raw compressed data to include additional metadata. We refer to this as *gzip compressed* data.

The Tcl commands related to these fall into three categories:

- Commands that operate on the entire data to be compressed or decompressed. These are discussed below in Section 8.5.1.
- Commands that operate in *stream* mode where the data is incrementally compressed or decompressed. These are described below in Section 8.5.2.
- Channel transforms where data is **transparently** compressed or decompressed during input-output operations are discussed in Section 21.2.4.
- Commands related to ZIP files are covered in Chapter 25.
- Utility commands for calculating checksums. These are discussed in Section 8.6.

All commands related to zlib compression are subcommands of the `zlib` command.

Since compression algorithms effectively target byte sequences, they must be passed binary strings, for example those directly constructed with the `binary format` (Section 8.3) command or by encoding strings with `encoding convertto` (Section 9.1.3) command. Do **not** directly pass arbitrary Tcl strings that are not binary strings.

8.5.1. Compressing strings

The commands discussed in this section expect the binary string that is to be operated on to be provided in a single argument. Let us create such a string to use for our examples.

```
% set bin [encoding convertto utf-8 [string repeat abcd 200]]
```

We will learn all about the `encoding` command in Chapter 9 but for now all you need to know is that the above creates a binary string.

[1] https://tools.ietf.org/html/rfc1951
[2] https://tools.ietf.org/html/rfc1950
[3] https://tools.ietf.org/html/rfc1952

8.5.1.1. Raw DEFLATE compression: zlib deflate|inflate

```
zlib deflate BINSTRING ?LEVEL?
zlib inflate COMPRESSED ?BUFFERSIZE?
```

The first pair, `zlib deflate` and `zlib inflate`, implement compression and expansion respectively using the raw DEFLATE algorithm of RFC 1951 without any metadata.

The *`LEVEL`* argument should be a number between `0` and `9` with a level of `0` indicating no compression and `9` indicating maximal compression, at the cost of performance. The default value is `1`.

```
bin2hex [set zbin [zlib deflate $bin]] → 4b 4c 4a 4e 49 1c c5 a3 78 14 63 c5 00
```

(Our repeated input string results in excellent compression!)

The inverse command is `zlib inflate`.

```
% encoding convertfrom utf-8 [zlib inflate $zbin]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcda...
```

When uncompressing, the `inflate` command will grow the buffer required for the uncompressed data as required. As a performance optimization, you can specify *`BUFFERSIZE`* as the expected length of data so that memory reallocations are avoided.

```
% encoding convertfrom utf-8 [zlib inflate $zbin 1000]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcda...
```

8.5.1.2. Zlib compression: zlib compress|decompress

```
zlib compress BINSTRING ?LEVEL?
zlib decompress COMPRESSED ?BUFFERSIZE?
```

A second set of commands, `zlib compress` and `zlib decompress`, implement compression and expansion respectively using the Zlib compressed data format defined in RFC 1950. This uses the same algorithm as `zlib deflate` but includes additional meta information. The well known `zip` compressed files primarily use this format.

The optional *`LEVEL`* and *`BUFFERSIZE`* parameters are as described above for the `zlib deflate` (Section 8.5.1.1) and `zlib inflate` (Section 8.5.1.1) commands respectively.

```
% bin2hex [set zbin [zlib compress $bin]]
→ 78 9c 4b 4c 4a 4e 49 1c c5 a3 78 14 63 c5 00 aa 4f 33 e0
% encoding convertfrom utf-8 [zlib decompress $zbin]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcda...
```

Notice from the output of the `bin2hex` command that the Zlib compression format contains within it the output of the raw DEFLATE data from the previous section.

8.5.1.3. Gzip compression: zlib gzip|gunzip

```
zlib gzip BINSTRING ?-level LEVEL? ?-header HDRDICT?
zlib gunzip COMPRESSED ?-headerVar VARNAME?
```

The last set of commands in this category `zlib gzip` and `zlib gunzip` are also based on the DEFLATE algorithm but this time using the format defined in RFC 1952. The popular `gzip` and `gunzip` command line utilities that produce `.gz` files use this format.

The syntax of these commands differs slightly from their brethren because the format supports more metadata values. The `-level` option serves the same purpose as the *`LEVEL`* argument in `zlib deflate` (Section 8.5.1.1) except it is supplied as an option switch instead of a plain argument. The `-header` option allows the caller to supply the associated metadata. *`HDRDICT`* should be a dictionary containing zero or more of the keys shown in Table 8.5.

Table 8.5. Gzip header keys

Key	Value
`comment`	A comment to be included in the Gzip metadata.
`crc`	A boolean value. If `true`, the GZIP header CRC is computed. This should be `false` for interoperability with the `gzip` program.
`filename`	Name of the file that was the source of the data.
`os`	Operating or file system type code as defined in RFC 1952. Common ones are `0` for FAT, `3` for Unix, `11` for NTFS.
`time`	Last modified time of the file as per `file mtime` (Section 12.2.2).
type	One of `binary` or `text` indicating the type of data being compressed.

Correspondingly, if the `-headerVar` option is used with the `zlib gunzip` command, the metadata values retrieved from the compressed data are stored in the variable *`VARNAME`* in the caller's context. The data is stored as a dictionary which may contain the same keys shown in Table 8.5 and an additional key, `size`, that contains the size of the compressed data.

This is illustrated in the example below.

```
% set hdr [list time [clock seconds] comment "A demo file"]
→ time 1743694528 comment {A demo file}
% bin2hex [set zbin [zlib gzip $bin -header $hdr]]
→ 1f 8b 08 10 c0 aa ee 67 00 00 41 20 64 65 6d 6f 20 66 69 6c 65 00 4b 4c 4a 4e...
% encoding convertfrom utf-8 [zlib gunzip $zbin -headerVar hdr2]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcda...
% print_dict $hdr2
→ comment   = A demo file
  crc       = 0
  os        = 0
  size      = 800
  time      = 1743694528
...Additional lines omitted...
```

8.5.2. Compressing streams

The commands discussed in the previous section all work in a “single-shot” manner where all the data that is to be operated on is provided in one call. This is neither convenient nor performant in terms of memory usage when the data becomes available in a discrete or piecemeal fashion. For such cases, Tcl provides the `zlib stream` command where data can be fed into the compression engine in incremental fashion.

The sequence of commands for compression and decompression are the same except for the algorithm passed (zip versus unzip etc.). A zlib stream is created and uncompressed/compressed data written to it in chunks and compressed/uncompressed data read back.

8.5.2.1. Creating a zlib stream: zlib stream

```
zlib stream ENGINE ?OPTIONS?
```

The `zlib stream` command creates a zlib stream. The *`ENGINE`* parameter is one of `deflate`, `inflate`, `compress`, `decompress`, `gzip` or `gunzip` and corresponds to the compression and decompression commands described in the preceding sections. The command returns a zlib streaming object to which data can be written and read.

Table 8.6 shows the options that can be used with the various engines.

Table 8.6. Zlib stream options

Option	Description
`-dictionary` *`BINDATA`*	Specifies a compression dictionary to be used for compressing or decompressing. *`BINDATA`* is a binary string and is not to be confused with a Tcl dictionary. See the explanation of the preset dictionary in RFC 1950[4]. This option can be used with the `deflate`, `inflate`, `compress` and `decompress` engines.
`-header` *`HEADER`*	Specifies the Gzip format metadata header. This option can only be used with the `gzip` engine.
`-level` *`LEVEL`*	Specifies the compression level. This option can be used with the `deflate`, `compress` and `gzip` engines.

Let us create two streaming objects to do Gzip compression and decompression that we will use in later sections.

```
% set compressor [zlib stream gzip -header {comment "A zlib demo"}]
→ ::tcl::zlib::streamcmd_1
% set decompressor [zlib stream gunzip]
→ ::tcl::zlib::streamcmd_2
```

[4] https://tools.ietf.org/html/rfc1950

8.5.2.2. Writing to a zlib stream

```
STREAM put ?OPTIONS? BINDATA
```

The stream object's `put` method writes data to a zlib stream. The command supports the options shown in Table 8.7.

Table 8.7. Zlib stream put options

Option	Description
`-dictionary BINDICT`	Sets `BINDICT` as the compression dictionary as described in Table 8.6.
`-finalize`	The use of this option is described in Section 8.5.2.3.
`-flush`	The use of this option is described in Section 8.5.2.9.
`-fullflush`	The use of this option is described in Section 8.5.2.9.

Note that only one of `-finalize`, `-flush` or `-fullflush` may be specified.

Multiple `put` commands may be invoked to add data in incremental fashion.

```
% $compressor put [encoding convertto utf-8 "abcd"]
% $compressor put [encoding convertto utf-8 "efgh"]
```

8.5.2.3. Finalizing a zlib stream: finalize, put -finalize

```
STREAM finalize
STREAM -put -finalize BINDATA
```

After all data has been written to the stream, it needs to be notified to complete the compression process and write out metadata. This can be done in one of two ways:

You can call the `finalize` method.

```
$compressor finalize
```

Alternatively, If you know the data you are writing is the last piece, you can pass the `zlib streams` option to the `put` method.

```
% $compressor put -finalize [encoding convertto utf-8 "ijkl"]
```

8.5.2.4. Reading from a zlib stream: get

```
STREAM get ?COUNT?
```

The `get` method reads back data written to a zlib stream.

If the *COUNT* argument is specified, the command returns that many bytes from the stream. If unspecified, all remaining data in the stream is returned.

```
set compressed_1 [$compressor get 2]
set compressed_remaining [$compressor get]
bin2hex $compressed_1 $compressed_remaining
→ 1f 8b 08 10 00 00 00 00 00 00 41 20 7a 6c 69 62 20 64 65 6d 6f 00 4b 4c 4a 4e...
```

Having seen the basic commands for zlib streams, here is a short demo that mirrors our previous examples from Section 8.5.1.

```
set strm [zlib stream deflate] ❶
for {set i 0} {$i < 200} {incr i} {
    $strm put [encoding convertto utf-8 "abcd"] ❷
}
$strm finalize ❸
set zbin [$strm get] ❹
$strm close ❺
bin2hex $zbin
→ 4b 4c 4a 4e 49 1c c5 a3 78 14 63 c5 00
```

❶ Create a streaming compressor object.
❷ Feed it binary data to be compressed.
❸ Indicate end of data.
❹ Retrieve compressed data.
❺ Release resources.

8.5.2.5. Computing zlib stream checksum

```
STREAM checksum
```

A zlib compression stream keeps track of the checksum of the uncompressed data written to it. This can be retrieved at any point with the `checksum` method.

```
% $compressor checksum
→ 4135066404
```

8.5.2.6. Reusing a zlib stream: reuse

```
STREAM reset
```

The `reset` method can be called to reuse an existing stream for new data. The stream can then be used exactly as if it were a new stream opened with `zlib stream`. This is a little more efficient than a close and open sequence.

```
$compressor reset → (empty)
```

8.5.2.7. Closing a zlib stream: close

```
STREAM close
```

Once a zlib stream is no longer required, it must be released by calling the `close` method on the stream object.

```
$compressor close → (empty)
```

8.5.2.8. Decompression streams

The above examples illustrated compression of data using zlib streams. Decompression works exactly like compression except for the engine used. We can incrementally decompress by writing the compressed data to the stream with the `put` method.

```
% set decompressor [zlib stream gunzip]
→ ::tcl::zlib::streamcmd_4
% $decompressor put $compressed_1
% $decompressor put -finalize $compressed_remaining
```

In the case of Gzip format, we can retrieve the metadata header for the compressed data with the `header` method.

```
% print_dict [$decompressor header]
→ comment    =
  crc        = 0
  filename   =
  os         = 0
  type       = binary
```

As always, the decompressed data itself is obtained with the `get` method.

```
set decompressed [$decompressor get]
encoding convertfrom utf-8 $decompressed
→ abcdefghijkl
```

8.5.2.9. Flushing zlib streams

```
STREAM flush
STREAM fullflush
```

Because an explanation of flushing requires an understanding of how the DEFLATE algorithm works, we do not go into details but refer you to the article by Thomas Pornin[5]. From a

[5] http://www.bolet.org/~pornin/deflate-flush.html

practical point of view, flushing a zlib compression stream allows the decompressing end to correctly decompress the data even under certain error conditions.

- In the case of a “sync flush”, a decompressor can decompress data up to the point at which the compressor invoked the flush even in case of errors in writing subsequent data.
- In the case of a “full flush”, it additionally allows a decompressor to decompress data beyond the point of the flush even in case of errors in earlier bytes.

For a Tcl zlib compression stream, a sync flush or full flush is effected by calling the `flush` and `fullflush` methods respectively.

Alternatively, you can pass the `zlib streams` or `zlib streams` options to the `put` method.

Flushing incurs a cost in compression efficiency. Generally, its use is dictated by the upper layer protocols that make use of compression. For example, compression in HTTP entails no flushing as the content is sent as a single blob. Packet oriented protocols on the other hand may mandate flushing at packet boundaries.

8.6. Computing checksums: zlib adler32|crc32

```
zlib adler32 BINDATA ?INITDATA?
zlib crc32 BINDATA ?INITDATA?
```

The `zlib` command has two subcommands, `zlib adler32` and `zlib crc32`, for computing Adler-32 and CRC-32 checksums on a binary string.

`BINDATA` is the binary string whose checksum is to be computed. *`INITDATA`* is used to initialize the computation.

```
zlib adler32 [encoding convertto utf-8 abcd] → 64487819
zlib crc32 [encoding convertto utf-8 abcd]   → 3984772369
```

9

Globalization

> To speak another language is to possess a second soul.
>
> — Charlemagne

Up to this point, we have primarily focused our discussion and examples on English text. It is now time to explore Tcl's support for *globalization*, the ability for applications to work with other languages and regions. This chapter describes Tcl support for the various components of globalization:

- Ability to exchange text information with other applications and storage systems via multiple encoding schemes, irrespective of content language and alphabet.
- *Internationalization* support that allows an application to be written free of any assumptions of language or region.
- *Localization* support that enables presentation of information in user selected languages and formats (e.g. dates) **without any program modification**.

9.1. Character encoding

Although at the script level, Tcl strings are best thought of as an abstract sequence of Unicode code points (Section 4.1.1) (or characters, loosely speaking) when it comes to storing on disk or sharing data with other programs, these strings need to be converted to and from a physical sequence of bytes. The method by which this conversion is done is defined by an *encoding*.

For example, consider the encoding of Unicode code point sequence `U+004f U+006c U+00e1` which is the Portuguese word Olá. As a physical sequence of bytes in a file it may be stored as

- `4f 6c e1` (ISO8859-9)
- `4f 6c c3 a1` (UTF-8)
- `4f 00 6c 00 e1 00` (UTF-16LE)

amongst many other possible encodings.

Encoding schemes may be fixed length, with each code point converted to the same number of bytes, or they may be variable length where the different characters are encoded to byte sequences of different lengths. Some commonly used encodings are UTF-8, which is almost universally the encoding of choice in modern protocols, ISO8859-1, intended for Western European languages, ShiftJIS for Japanese and Big5 for Chinese.

This section describes Tcl facilities for directly converting to and from various encodings. Implicit encoding during channel I/O is covered separately in Section 13.12.

9.1.1. Encoding profiles

Several anomalous cases may arise in the process of converting Tcl strings to and from encoded byte sequences.

- The specific Unicode code point may not be representable in that particular encoding. For example, single byte encodings like ISO8859-1 can only represent 256 values out of the thousands of code points.
- Due to transmission or other errors, the byte sequence might be corrupted resulting in illegal byte values or truncated sequences in multi-byte encodings.
- The byte sequence is correct as per its source encoding but the application expects a different encoding, resulting in error conditions similar to the above.

In such cases, applications may want to take any one of several actions.

- Signal an error and stop further processing of the data.
- Substitute the invalid byte(s) and continue processing.
- Discard the invalid bytes and continue processing.

Appropriate handling depends on the application. A banking application necessarily needs to be strict in its handling of data and would likely signal an error. On the other hand, a XML viewer or browser may choose the second approach using a special character as a visual cue to the user that the particular character could not be decoded.

The above behaviours are selectable through *encoding profiles*. Profiles in Tcl 9.0 include:

- The `strict` profile implements the first action above, signalling an error. The error notification mechanism depends on whether the error occurred during I/O or during an explicit encoding command invocation. This profile is compliant with the Unicode standard and is the default profile.
- The `replace` profile implements the second action above. When decoding a byte sequence, the invalid byte(s) are replaced with the `U+FFFD REPLACEMENT CHAR` code point as recommended by the Unicode standard. When encoding a Tcl string, any code points not supported by the encoding are replaced with an encoding-dependent character, usually the question mark `?` character. This profile is also Unicode standard compliant.
- The `tcl8` profile also implements the second action but differs from the `replace` profile in that bytes during decoding are treated as though they had been encoded using ISO8859-1 encoding wherein each byte value is mapped to the numerically equivalent code point. This in essence has the effect of silently corrupting data and is not Unicode standard compliant. It is present only for reasons of backward compatibility and its use should be avoided.

Tcl does not have a profile that simply discards invalid bytes as this leads to security vulnerabilities and is strongly discouraged by the Unicode standard.

The use of profiles and differences in their behaviour will be seen through this chapter.

Encoding profiles are not available in Tcl 8.6 and earlier where encoding errors are always treated as in the `tcl8` profile.

Profiles may be added or removed in newer Tcl versions. The list of profiles supported can be obtained at runtime with the `encoding profiles` command.

```
encoding profiles → replace strict tcl8
```

9.1.2. Supported encodings: encoding names

```
encoding names
```

The `encoding names` command returns the list of encodings supported by a Tcl application. The `utf-8` and `iso8859-1` encodings are guaranteed to be always present.

```
% encoding names
→ cp860 cp861 cp862 cp863 tis-620 cp864 cp865 cp866 gb2312-raw gb12345 utf-32be...
```

9.1.3. Encoding characters: encoding convertto

```
encoding convertto ?-profile PROFILE? ?-failindex VARNAME? ?ENC? STRING
```

The `encoding convertto` command encodes a string. The result of the command is a binary string containing the sequence of bytes in the encoding *ENC*. Thus, assuming the variable `hello` contained the word Olá,

```
bin2hex [encoding convertto iso8859-9 $hello] → 4f 6c e1
bin2hex [encoding convertto utf-8 $hello]     → 4f 6c c3 a1
bin2hex [encoding convertto utf-16le $hello]  → 4f 00 6c 00 e1 00
```

The `-profile` and `-failindex` options control how failures are handled, for example if a character cannot be represented in the target encoding. Attempt to convert Olá as above to ASCII encoding will raise an error exception by default because á cannot be encoded in ASCII.

```
% encoding convertto ascii $hello
Ø unexpected character at index 2: 'U+0000E1'
```

A profile (Section 9.1.1) can be passed to the command via the `-profile` option to change this.

```
% encoding convertto -profile strict ascii $hello ❶
Ø unexpected character at index 2: 'U+0000E1'
% encoding convertto -profile replace ascii $hello ❷
→ Ol?
```

❶ This is the default profile.
❷ The `?` is the replacement character in ASCII.

The `-failindex` option allows the caller to encode as much data as possible while being notified of the point of failure. The name of a variable is passed as the option value. In the absence of errors, `-1` is stored to the variable. Otherwise, the variable holds the position of the failure in the source string. No exception is raised and the command returns the encoded byte sequence up to that point.

```
encoding convertto -failindex indexVar ascii abc    → abc
set indexVar                                        → -1
encoding convertto -failindex indexVar ascii $hello → Ol
set indexVar                                        → 2
```

The option only makes sense in conjunction with strict profiles as the other profiles do not raise an error in any case.

The `-profile` and `-failindex` options are not available in Tcl 8.6 and earlier.

9.1.4. Decoding characters: encoding convertfrom

```
encoding convertfrom ?-profile PROFILE? ?-failindex VARNAME? ENC BINARY
```

The `encoding convertfrom` command returns the Tcl string resulting from decoding an encoded sequence of bytes. This is the inverse of the `encoding convertto` command.

ENC is the encoding of the binary string *BINARY*. Like `encoding convertto`, `encoding convertfrom` will raise an exception on bytes that are not valid as per the specified encoding unless the `-profile` or `-failindex` options are passed.

The following examples illustrate decoding of the byte sequence (in hex) `61 e2 63` resulting from aβc encoded with the CP1253 (code page Greek on Windows).

```
set greek a\u03b2c                                          → aβc
bin2hex [set cp1253bytes [encoding convertto cp1253 $greek]] → 61 e2 63
encoding convertfrom cp1253 $cp1253bytes                    → aβc
```

This byte sequence is not valid as UTF-8 and attempts to decode it as UTF-8 using the (default) `strict` profile results in an error being notified, either via an exception or the `-failindex` option in the same manner we saw with `convertto` in the previous section.

```
% encoding convertfrom utf-8 $cp1253bytes
Ø unexpected byte sequence starting at index 1: '\xE2'
% encoding convertfrom -profile strict utf-8 $cp1253bytes
Ø unexpected byte sequence starting at index 1: '\xE2'
% encoding convertfrom -profile strict -failindex indexVar utf-8 $cp1253bytes
→ a
% set indexVar
→ 1
```

A different encoding profile can be passed through the `-profile` option to substitute the invalid bytes instead of raising an error.

```
encoding convertfrom -profile replace utf-8 $cp1253bytes → a�c
encoding convertfrom -profile tcl8 utf-8 $cp1253bytes    → aâc
```

Note with the Unicode standard compliant `replace` profile, the invalid characters are visually identifiable as `�`. With the `tcl8` profile on the other hand, there is no such visual clue to the user that the data is corrupted or invalid. This profile should therefore be avoided. It is only present for use by existing applications that intentionally or otherwise, treated such bytes as ISO8859-1 encoded bytes.

The `-profile` and `-failindex` options are not available in Tcl 8.6 and earlier.

9.1.5. Adding new encodings: encoding dirs

```
encoding dirs ?DIRLIST?
```

The encodings supported by a Tcl application are based on definitions in files read at runtime. These files are identified by their extension `.enc` and located in one of the directories in the list returned by the `encoding dirs` command.

If *`DIRLIST`* is not passed, the command returns the list of directories searched for encodings.

```
encoding dirs → c:/tcl/magic/lib/tcl9.0/encoding
```

You can also change the list of directories searched by passing an argument to `encoding dirs`. For example, to add a new directory containing additional encodings to the existing search path:

```
% encoding dirs [linsert [encoding dirs] end C:/my/extra/encodings]
→ c:/tcl/magic/lib/tcl9.0/encoding C:/my/extra/encodings
```

9.1.6. The system encoding: encoding system

```
encoding system ?ENC?
```

The system encoding is the encoding used for the following purposes:

- On platforms other than Windows, the system encoding is used to encode and decode strings passed to and from system calls that take string arguments. On Windows, Tcl always uses the Win32 wide character API and UTF-16 encodings irrespective of the system encoding.
- The system encoding is also used as the default encoding for I/O channels on all platforms until modified with the `chan configure` command.

Without any arguments, the command simply returns the current system encoding.

```
encoding system → utf-8
```

If *ENC* is passed, the system encoding will be changed to the encoding of that name. This should be done only after very careful consideration as it has impact across all system calls on non-Windows platforms.

9.1.7. Reading and writing encoded data

Tcl provides the ability to configure input and output streams to automatically convert from and to a specific encoding without having to explicitly invoke the `encoding` command. We defer this topic to Section 13.12.

9.2. Internationalization

Internationalization of an application involves adapting it to work independently of any specific language or alphabet. Tcl strings support the entire set of characters defined in the Unicode standard. Further, commands that work with strings are also fully Unicode aware so features like regular expressions, character classification (`string isdigit`) work with all languages. Thus Tcl applications are inherently internationalized without needing any additional considerations.

9.3. Localization

Localization of an application for a language and region requires

- Externalizing all user visible strings so they are not present in the program source code but instantiated at runtime based on the user's language.
- Formatting numbers, dates, currencies etc. as per the user's region.
- Sorting and comparisons of strings based on language and regional conventions.
- Layout of the user interface in terms of sizing, culture-sensitive icons and similar.

The last two have no direct support in Tcl. Sorting and comparison is strictly based on Unicode code points and user interface issues are outside the purview of Tcl.

Formatting of dates and times is covered separately in Chapter 11. Formatting of other value types, as decimal strings with separators for example, requires external packages.

The rest of this section covers Tcl's `msgcat` package which enables the first of the above requirements through message catalogs.

9.3.1. Locales

A *locale* is a container for a collection of settings and translations for a language and region. Locales are identified in Tcl by a locale string consisting of

- a language code as defined in international standard ISO 639,
- optionally followed by an `_` and a country code as defined in standard ISO 3166,
- optionally followed by an `_` and a system specific code.

For example, `en` identifies the generic English locale while `en_us` and `en_gb` identify the variations for USA and Great Britain respectively. Locales form a hierarchy so that if a translation is not found in `en_us`, it is looked up in `en` and then in the top of the hierarchy, called the *ROOT* locale.

When an application starts, the initial locale is set based on the values of the `LC_ALL`, `LC_MESSAGES` and `LANG` environment variables in that order. On Windows, if none of these are defined, the locale is retrieved from the registry. As a last resort, it is set to `C`.

9.3.2. Message catalogs: mcset, mcmset, mcflset, mcflmset

```
msgcat::mcset LOCALE KEY ?LOCALIZEDSTRING?
msgcat::mcmset LOCALE LOCALIZATIONLIST
msgcat::mcflset KEY ?LOCALIZEDSTRING?
msgcat::mcflmset LOCALIZATIONLIST
```

A *message catalog* contains translations for each locale keyed by the locale name. It is stored in multiple files, one per locale, in a single directory. The file names have the form `LOCALE.msg` where `LOCALE` is a lower case string identifying the locale. Thus the file `es.msg` will store Spanish translations. As a special case, the translations for the ROOT locale are stored in the file `ROOT.msg`. The content of message files must be in UTF-8 encoding.

Each application or package will normally store all its localization files within a single application or package specific directory. The Tcl core localization files are in the `msgs` subdirectory of the directory path stored in the `tcl_library` variable.

Within each file, the localization strings for that locale are defined using one of the commands listed above from the `msgcat` namespace.

Each command defines the localized string corresponding to one or more *keys* within a locale. If `LOCALIZEDSTRING` is not specified, it defaults to `KEY` itself.

Four message catalog files illustrating their use for locales `en_us`, `en`, `fr` and `de` corresponding US English, English, French and German are shown below. The first of these uses `mcset` which maps a single key for the specified locale.

```
# en_us.msg
msgcat::mcset en_us hello "Wassup world!"
```

The `mcmset` command is more convenient for defining multiple strings. Its *LOCALIZATIONLIST* argument is a list of alternating keys and localized strings.

```
# en.msg
msgcat::mcmset en {
    hello "Hello world!"
    goodbye "Goodbye cruel world!"
    TIME "The current time is %s"
}
```

The `mcflset` and `mcflmset` commands are analogous to `mcset` and `mcmset` respectively except their target locale is implicitly defined based on the name of the containing file. They can only be used inside of message catalog files loaded with `mcload` (Section 9.3.3).

```
# fr.msg
msgcat::mcflset hello "Bonjour le monde!"
msgcat::mcflset goodbye "Adieu monde cruel!"
msgcat::mcflset TIME "L'heure actuelle est %s"
```

```
# de.msg
msgcat::mcflmset {
    hello "Hallo Welt!"
    goodbye "auf Wiedersehen, grausame Welt!"
}
```

9.3.3. Loading message catalogs: mcload, mcloadedlocales

```
msgcat::mcload MSGCATDIR
msgcat::mcloadedlocales loaded
msgcat::mcloadedlocales clear
```

To use message catalogs, the `msgcat` package which contains commands related to localization must be loaded first followed by the message catalog itself. The latter is loaded with the `msgcat::mcload` command.

Message catalog files may be stored in any directory but it is common to store them in a subdirectory under the package's script directory. They can be then loaded as

```
package require msgcat                                    → 1.7.1
msgcat::mcload [file join [file dirname [info script]] msgs] → 2
```

Loaded locales can be managed with the `mcloadedlocales` command ensemble. The `loaded` subcommand returns the list of locales that have been loaded. The `clear` subcommand unloads locales that are not currently in the locale preferences (Section 9.3.7).

```
msgcat::mcloadedlocales loaded → en_us en {}
```

9.3.4. Retrieving translations: mc

```
msgcat::mc KEY ?ARG …?
```

The final step in localization is replacement of literal strings with translated equivalents obtained with the `mc` command. Our Hello World application would become

```
puts [msgcat::mc hello] → Wassup world!
```

The literal greeting *Hello world* is replaced by the call to `mc`. The passed key *hello* is looked up in the current locale which happens to be `en_us`.

The argument *`KEY`* is looked up in the message catalog. If no entry is found, the key itself, `Hi there!` in the example below, is returned.

```
msgcat::mc "Hi there!" → Hi there!
```


Although it is convenient to use the English (or any language for that matter) localization as the key so as to not have to define entries in the message catalog, our examples use explicit definitions for illustrative purposes. Also note that returning the key on lookup failures relies on the default behaviour of the `mcunknown` commmand (Section 9.3.9) and that may be changed by an application or package.

Any additional arguments to `mc` are passed to the `format` command (Section 4.20) along with the localized string and its result returned. For example, using the definition of `TIME` in our message catalogs shown earlier, we can print the current time as

```
set now [clock format [clock seconds] -format %T] → 21:05:28
msgcat::mc TIME $now                              → The current time is 21:05:28
```

The second line above is roughly equivalent to

```
set fmt [msgcat::mc TIME] → The current time is %s
format $fmt $now          → The current time is 21:05:28
```

The above example also illustrates another feature of message catalogs — the locale hierarchy. The key `TIME` is not defined in the `en_us` locale and was therefore looked up in the parent `en` locale.

9.3.5. Comparing translation lengths

```
msgcat::mcmax ?KEY …?
```

In user interfaces, it is sometimes useful to know the longest length of displayed strings, for example to set up equal column widths in a table. The `mcmax` command targets this use case.

The command returns the longest length amongst the translations for the passed arguments in the current locale.

```
msgcat::mcmax hello goodbye → 20
```

9.3.6. Retrieving and setting the locale: mclocale

```
msgcat::mcutil getsystemlocale
msgcat::mclocale ?LOCALE?
```

When the `msgcat` package is loaded it sets the current locale based on system settings (Section 9.3.1). This default locale can be retrieved with `mcutil getsystemlocale`.

```
msgcat::mcutil getsystemlocale → en_us
```

There are cases where the default locale is not appropriate. For example, a web server should set the locale based on the client's language and region, not its own. The `mclocale` command can be used to retrieve or change the locale.

In the absence of any arguments, the command returns the locale. If *`LOCALE`* is specified, the locale is set accordingly and returned. The effect of the command is seen below.

```
set oldLocale [msgcat::mclocale] → en_us
msgcat::mc hello                 → Wassup world!
msgcat::mclocale en_gb           → en_gb
msgcat::mc hello                 → Hello world!
msgcat::mclocale $oldLocale      → en_us
```

In the case of an application running multiple Tcl interpreters, the `mclocale` only changes the locale for the interpreter in which the command is invoked. We will discuss multiple interpreters in Chapter 23.

Setting the locale with `mclocale` also impacts the locale preferences described next.

9.3.7. Locale preferences: mcpreferences

```
msgcat::mcpreferences ?LOCALE …?
msgcat::mcutil getpreferences LOCALE
```

Setting a locale actually sets up preference order of locales to be searched when looking up a key. This preference order can be retrieved and modified with the `mcpreferences` command.

If no arguments are passed, the command returns the current order of locale preferences. If any arguments are specified, the order is set accordingly.

```
msgcat::mcpreferences             → en_us en {}
msgcat::mcpreferences en_us fr {} → en_us fr {} ❶
msgcat::mc goodbye                → Adieu monde cruel! ❷
```

❶ Change the order so `fr` replaces `en`.
❷ Now keys not present in `en_us` are first looked up in `fr`.

When arguments are supplied, the command also sets the current locale to be the first argument passed. Conversely, calling `mclocale` also modifies the order of preferences.

```
msgcat::mcpreferences fr de {} → fr de {}
msgcat::mclocale               → fr
msgcat::mclocale en_us         → en_us
msgcat::mcpreferences          → en_us en {}
```

The `mcutil getpreferences` command retrieves the preferences for a particular locale.

This does not change the locale itself. It is useful for setting up custom preferences; for example to construct the preferences that includes `en_us` and `fr` along with their parents.

```
msgcat::mcpreferences {*}[lreplace \
                            [msgcat::mcutil getpreferences en_us] end end \
                            {*}[msgcat::mcutil getpreferences fr]]
msgcat::mcpreferences
→ en_us en fr {}
```

9.3.8. Partitioning catalogs with namespaces: mcn

```
msgcat::mcn NAMESPACE KEY ?ARG …?
```

One issue that can arise when independent packages have their own message catalogs is the potential for conflict between the keys used by each package. The use of namespaces in message catalogs addresses this issue. Readers unfamiliar with namespaces may want to return to this section after reading Chapter 16.

When localized strings are retrieved with the `mc` command (Section 9.3.4), it looks up the message catalog within the context of the namespace from which it is called. Accordingly, packages that make use of the message catalogs should invoke the `mcset` family of definition commands from within the package's namespace.

The English localization for our anniversary package may contain

```
namespace eval ::anniversary {
    msgcat::mcset en greeting "Happy anniversary!"
}
→ Happy anniversary!
```

Similarly, the Christmas greetings package localization file contains

```
namespace eval ::xmas {
    msgcat::mcset en greeting "Merry Christmas!"
}
→ Merry Christmas!
```

These files define the greeting message based on the occasion as reflected by the containing namespace. The printed greeting as shown below will then depend on the namespace context in which the `mc` command is invoked.

```
msgcat::mclocale en_us                                → en_us
msgcat::mc greeting                                   → greeting ❶
namespace eval ::anniversary {msgcat::mc greeting}    → Happy anniversary!
namespace eval ::xmas {msgcat::mc greeting}           → Merry Christmas!
```

❶ Prints `greeting` since the global namespace has no catalog entry for `greeting`.

Alternatively, the `mcn` command can be used in place of `mc` to explicitly specify a namespace. The first argument to the command is the target namespace. Remaining arguments are as for the `mc` command.

```
msgcat::mcn ::anniversary greeting → Happy anniversary!
msgcat::mcn ::xmas greeting       → Merry Christmas!
```

If a namespace does not define a message catalog entry that matches the locale, ancestor namespaces are searched as well. So if the `anniversary` namespace had a child namespace `gold`, the following would work.

```
namespace eval ::anniversary::gold {msgcat::mc greeting} → Happy anniversary!
```

On failing to find a `greeting` entry in any suitable locale in the `anniversary::gold` namespace, the `mc` command would check `anniversary` and the global namespaces in turn.

9.3.9. Unknown message keys: mcunknown, mcexists

```
msgcat::mcunknown LOCALE KEY ?ARG …?
msgcat::mcexists ?-exactnamespace? ?-exactlocale? ?-namespace NS? KEY
```

When the `mc` command (Section 9.3.4) does not find a localization defined in the current locale, it invokes the `msgcat::mcunknown` procedure and returns its value. The default definition of `mcunknown` simply returns the passed lookup key. An application can redefine this command to take some other action it wishes.

Applications can check whether an entry exists for a key with the `mcexists` command. The command returns `1` if the lookup of *KEY* would be successful and `0` otherwise. By default, the lookup is done as in the `mc` command.

```
msgcat::mcexists hello      → 1
msgcat::mcexists doesnotexist → 0
```

The `-exactlocale` option modifies the check so that the command only checks the current locale and not any ancestors.

```
msgcat::mcexists -exactlocale hello   → 1
msgcat::mcexists goodbye              → 1
msgcat::mcexists -exactlocale goodbye → 0
```

The existence check is normally done in the namespace context in which `mcexists` is invoked. The `-namespace` option can be passed to do the check from a different namespace. Further, `-exactnamespace` prevents lookup in ancestor namespaces.

```
msgcat::mcexists -namespace ::anniversary::gold greeting                → 1
msgcat::mcexists -exactnamespace -namespace ::anniversary::gold greeting → 0
```

9.3.10. Private package locales

The `msgcat` packages provides additional commands as a convenience for packages that want to define message catalogs within their own namespace. This section only provides a summary as their use is analogous to the preceding discussion on global and namespace-specific catalogs. In particular, the implementation is based on the latter using the package namespace. The primary advantage of using the commands in this section over the direct use of the underlying namespace-specific catalogs is that the commands will also work in OO class and object methods within the package namespace.

9.3.10.1. Managing package locales: mcpackagelocale

```
msgcat::mcpackagelocale clear
msgcat::mcpackagelocale get
msgcat::mcpackagelocale isset
msgcat::mcpackagelocale loaded
msgcat::mcpackagelocale preferences ?LOCALE …?
msgcat::mcpackagelocale present
msgcat::mcpackagelocale set ?LOCALE?
msgcat::mcpackagelocale unset
```

The first of these commands is `mcpackagelocale` which is a command ensemble.

The `msgcat::mcpackagelocale clear` command deletes any locales that are loaded for the package but not present in the package preferences.

The `msgcat::mcpackagelocale get` command returns the locale in effect for the package.

The `msgcat::mcpackagelocale isset` command returns `1` if a private locale is set for the package, else `0`.

The `msgcat::mcpackagelocale loaded` command returns list of locales loaded for this package.

The `msgcat::mcpackagelocale preferences` command sets the package locale preferences to specified arguments and the package locale to the first argument. If no arguments are specified, returns the current preferences for the package, private if set and global otherwise.

The `msgcat::mcpackagelocale present` command returns `1` if *`LOCALE`* has been loaded for the package, else `0`.

The `msgcat::mcpackagelocale set` command sets the locale for the package. If *`LOCALE`* is not specified, the locale is copied from the global locale.

The `msgcat::mcpackagelocale unset` command deletes the private locale for the package, reverting back to using the global profile.

9.3.10.2. Package locale options: mcpackageconfig

```
msgcat::mcpackageconfig get OPTION
msgcat::mcpackageconfig isset OPTION
msgcat::mcpackageconfig set OPTION VALUE
msgcat::mcpackageconfig unset OPTION
```

The private package locale facility also supports options shown in Table 9.1. These are managed with the `mcpackageconfig` command ensemble.

The `msgcat::mcpackageconfig get` command returns the value of the specified option.

The `msgcat::mcpackageconfig isset` command returns `1` if the specified option has been set, else `0`.

The `msgcat::mcpackageconfig set` command sets the value of the specified option.

The `msgcat::mcpackageconfig unset` command unsets the specified option.

Table 9.1. Package locale options

Option	Description
changecmd	If set, this should be a command callback to be invoked when the default locale is changed.
loadcmd	If set, this should be a command callback to be invoked to load message files.
mcfolder	The directory containing the message folder for the package.
unknowncmd	If set, this is the command to be used for the package in lieu of the global `mcunknown` command.

9.3.10.3. Package namespace: mcpackagenamespaceget

The namespace associated with a package can be obtained with the `mcpackagenamespaceget` command. It takes no arguments and command differs from similar commands described in Chapter 16 in that when called from OO class and object methods, it returns the namespace containing the class and object definitions as opposed to the namespace for the class or object itself.

```
namespace eval mcdemo {
    oo::class create C {
        method demo {} {
            puts [namespace current]
            puts [msgcat::mcpackagenamespaceget]
        }
    }
}
mcdemo::C create obj
obj demo
namespace delete mcdemo
→ ::oo::Obj532
  ::mcdemo
```

Notice the difference between the two output lines.

All private locale state and settings for a package can be reset with the `mcforgetpackage` command. The command does not take arguments and resets the state of the calling package.

For further details related to private package locales and the above commands, see the Tcl `msgcat` reference documentation.

9.4. Internationalized Domain Names: tcl::idna

```
tcl::idna decode HOSTNAME
tcl::idna encode HOSTNAME
tcl::idna puny decode STRING ?NOCASE?
tcl::idna puny encode STRING ?NOCASE?
tcl::idna version
```

The **Internationalized Domain Names in Applications** (IDNA) standard defines an ASCII encoding of non-ASCII Internet DNS names. The `tcl::idna` package provides commands to convert strings to and from this encoding.

```
package require tcl::idna → 1.0.1
tcl::idna version         → 1.0.1
```

The `tcl::idna encode` and `tcl::idna decode` commands encode and decode strings. The strings passed must be consist of characters that are valid in DNS names.

```
% tcl::idna encode aßc
→ xn--ac-gia
% tcl::idna decode xn--ac-gia
→ aßc
% tcl::idna encode slash/is/not/valid/in/dns
Ø bad character "/" in DNS name
```

IDNA uses an encoding scheme known as punycode. The `tcl::idna puny` command provides access to this scheme. Note that the encoding scheme itself does not care about validity of characters in DNS.

```
% tcl::idna puny encode aßc
→ ac-gia
% tcl::idna puny decode ac-gia
→ aßc
% tcl::idna puny encode punycode/does/not/care/slash/invalid/in/DNS
→ punycode/does/not/care/slash/invalid/in/DNS-
```

10

Regular Expressions

Like the glob patterns we saw earlier, *regular expressions* (**RE**) define search patterns where certain characters, termed *metacharacters*, have a special meaning. Compared to glob patterns, regular expressions are both considerably more powerful and more complex.

For those are familiar with regular expressions in other languages, note that the Tcl implementation differs slightly in syntax, particularly for advanced features.

In fact, Tcl itself supports three forms of regular expressions. Basic, Extended and Advanced. Only the last of these is described here as it is almost completely a superset of the others.

Table 10.1 summarizes the metacharacters used in RE's. Their use is detailed in later sections describing the commands related to regular expressions.

Table 10.1. Basic regular expression syntax

Character	Description
`^`	Matches the beginning of the string.
`$`	Matches the end of the string.
`.`	Matches any single character.
`[…]`	Matches any character from the set within the brackets.
`\`	Acts as an escape to assign special meaning to the next character or treat a metacharacter as a literal.
`[a-z]`	Matches any character in the range a..z.
`[^…]`	Matches any character *not* in the set given.
`(…)`	Groups a pattern into a sub-pattern.
p`|`*q*	Matches pattern *p* or pattern *q*.
`*`	Matches 0 or more occurrences of the preceding pattern.
`+`	Matches 1 or more occurrences of the preceding pattern.
`?`	Matches 0 or 1 occurrences of the preceding pattern.
`{`*n*`}`	Matches exactly *n* occurrences of the preceding pattern.
`{`*n*`,`*m*`}`	Matches between *n* and *m* occurrences of the preceding pattern.

10.1. Matching regular expressions: regexp

```
regexp ?options? RE STRING ?MATCHVAR MATCHVAR . . .?
```

The `regexp` command is used to match a string against a RE. In its simplest form, with no options or optional arguments specified, the command returns `1` if *`STRING`* matches the regular expression *RE* and `0` otherwise.

10.1.1. Matching specific characters

A character in *RE* that is not a metacharacter will match itself in *`STRING`*. Thus to look for the sequence `XY` in a string,

```
regexp XY aaXYbb → 1
regexp XY aaYXbb → 0
```

Note that in the absence of anchors (Section 10.1.7.1), it suffices for any substring of *`STRING`* to match the expression.

Character escapes

Certain characters that do not have a printable representation or are otherwise difficult to include in text can be included via an escape sequence prefixed with a backslash (\). For example, the newline character is represented by the sequence `\n` and Unicode characters can be represented as `\uhhhh` or `\Uhhhhhhhh` sequences (where `h` is a hexadecimal digit). See the documentation for `re_syntax` in the Tcl command reference for a list of character escape sequences.

The processing of these backslash sequences is in addition to any backslash substitution that might be done by the Tcl parser. Thus the following two commands are equivalent.

```
regexp "\\t" abc\tdef → 1
regexp {\t} abc\tdef  → 1
```

In the first case, the Tcl parser converts the `\\` sequence to a single `\` so the `regexp` command sees the argument as `\t`. In the second case, the enclosing braces prevent the Tcl parser from any backslash processing and again the `regexp` command sees `\t`.

Backslashes are also used for purposes other than character escapes. We will see these as we go along.

10.1.2. Matching any character

The metacharacter period (`.`) in a *RE* matches *any* character. For example, `X.Y` will match substrings containing an `X` and `Y` separated by exactly one character.

```
regexp X.Y aXcYb  → 1
regexp X.Y aXYb   → 0
regexp X.Y aXccYb → 0
```

10.1.3. Bracket expressions and character classes

We have already seen that characters in *RE* that are not metacharacters are matched against themselves in the string. If instead of matching any character, we wanted to match any of a **set** of characters, we can specify them as a *bracket expression* by enclosing them in `[]`.

```
regexp {ab[XYZ]cd} abYcd  → 1 ❶
regexp {ab[XYZ]cd} abQcd  → 0 ❷
regexp {ab[XYZ]cd} abXYcd → 0 ❸
```

❶ Y is in the bracket expression
❷ `Q` is *not* in the bracket expression
❸ `XY` is not a single character

The *RE* in the above example is enclosed in braces because the characters `[]` have special meaning to both the Tcl parser as well as RE syntax. Enclosing them in braces ensures they will not be treated as special characters by the Tcl parser. Because there are several other characters such as `$` and `\` that are treated specially by both the parser and RE, it is generally a good idea to enclose the RE in braces in all but the simplest cases.

A bracket expression has its own set of special character sequences described below and most RE metacharacters like `.`, `*` and `?` are treated as normal characters within the brackets. Notice in the next example how `.` loses its metacharacter status when placed within a bracket expression.

```
regexp {a.c} abc   → 1
regexp {a[.]c} abc → 0
regexp {a[.]c} a.c → 1
```

An expression of the form *`X-Y`* within the brackets includes all characters between *`X`* and *`Y`*. For example `a-z` includes all lower case English alphabetic characters, `0-9` includes all Western Arabic numerals and so on.

```
regexp {[0-9]} abc  → 0
regexp {[0-9]} a5c  → 1
regexp {[-0-9]} a-c → 1 ❶
```

❶ To include `-`, specify it as the first character

In the above examples, we are matching a single character which may be present anywhere in the string.

Another way to specify characters in bracketed expressions involves *character classes* of the form `[:`*`CLASSNAME`*`:]` where *`CLASSNAME`* is a name denoting a predefined set of characters. Tcl defines the character classes shown in Table 10.2.

Table 10.2. Regular expression character classes

Class	Description
`[:alnum:]`	Alphanumeric character
`[:alpha:]`	A letter
`[:blank:]`	Space or tab character
`[:cntrl:]`	Control character
`[:digit:]`	Decimal digit
`[:graph:]`	A character with a graphical representation
`[:lower:]`	Lower case letter
`[:print:]`	Printable character (same as `graph` plus the space character)
`[:punct:]`	Punctuation character
`[:space:]`	White space character
`[:upper:]`	Upper case letter
`[:xdigit:]`	Hexadecimal digit

Our previous examples for digits could also be more generally written as below. Note the doubled `[[]]`, the outermost set indicating a bracket expression and the inner set specifying a named character class.

```
regexp {[[:digit:]]} abc      → 0
regexp {[[:digit:]]} a5c      → 1
regexp {[[:digit:]]} a\u0968c → 1
```

The last example above shows that, unlike the `[0-9]` RE used earlier, the `[:digit:]` class is not restricted to ASCII and matches the Unicode code point U+0968 which is the Devanagari 2.

A bracket expression can include multiple characters, character ranges and classes concatenated together to indicate a “inclusive-or” combination. The RE below will match a string beginning with `a`, followed by any of the characters `x`, `y`, an upper case letter or digit, and ending with `b`.

```
regexp {a[xy[:upper:][:digit:]]b} axb → 1
regexp {a[xy[:upper:][:digit:]]b} a5b → 1
regexp {a[xy[:upper:][:digit:]]b} aQb → 1
regexp {a[xy[:upper:][:digit:]]b} aqb → 0
```

If a bracket expression starts with `^`, it matches characters **not** in the set.

```
regexp {a[xy[:digit:]]b} a5b  → 1
regexp {a[^xy[:digit:]]b} ayb → 0
regexp {a[^xy[:digit:]]b} a5b → 0
regexp {a[^xy[:digit:]]b} aQb → 1
```

Tcl regular expressions also support shortands, prefixed with `\`, for some commonly used classes. These are shown in Table 10.3.

Table 10.3. Character class shorthands

Shorthand	Bracket expression	Description
`\d`	`[[:digit:]]`	Digit
`\D`	`[^[:digit:]]`	Non-digit
`\s`	`[[:space:]]`	White space
`\S`	`[^[:space:]]`	Non-white space
`\w`		Alphanumerics and characters considered as part of words. See reference documentation.
`\W`		Complement of `\w`

For example, using `\d` in lieu of `[:digit:]`, expression.

```
regexp {a\db} a5b → 1
regexp {a\Db} a5b → 0
```

The `\d`, `\s` and `\w` shorthands can be used inside of bracketed expressions as well but the inverse versions of these, `\D`, `\S` and `\W`, cannot and you have to use the `^` prefix instead.

```
regexp {a[\d\s]b} a5b  → 1
regexp {a[\d\s]b} a\tb → 1
regexp {a[^\d\s]b} a5b → 0
```

10.1.4. Atoms and Groups

An atom is a single character in any of the forms described earlier (literal character, character escape or character class) or a group. Thus in the RE below, the components `a`, `[[:digit:]]` and `\n` are all atoms.

```
a[[:digit:]]\n
```

Subexpressions within a RE can be grouped with parenthesis. This treats the contents within the parenthesis as a single atom to which quantifiers (Section 10.1.5), alternation (Section 10.1.6) and such can be applied. In the first line in the example below, the `+` quantifier only applies to `Y` while in the second it applies to `XY`.

```
regexp {aXY+b} aXYXYb   → 0
regexp {a(XY)+b} aXYXYb → 1
```

Groups as used above use *capturing* parenthesis in that the string matching the subexpressions within parenthesis can be used in back references (Section 10.1.8) and substring extraction (Section 10.1.10).

An alternate form of grouping uses *non-capturing* parenthesis specified as `(?:RE)` where the leading left parenthesis is followed immediately by `?:`. The equivalent non-capturing version of our example above would be

```
regexp {a(?:XY)+b} aXYXYb → 1
```

The difference from capturing parenthesis is that in this case the substring matching the *RE* expression is not accessible via back references and cannot be extracted.

We will see examples and use of these forms in later sections.

10.1.5. Quantifiers

Quantifiers are appended to an atom to specify how many *consecutive* occurrences of that atom would be considered a match. For example, the expression `a+` would match one or more consecutive occurrences of the character `a`.

The various forms of quantifiers are shown in Table 10.4.

Table 10.4. Regular expression quantifiers

Quantifier	Description	Example

`*`	Matches 0 or more occurrences of the atom	`regexp {aX*b} ab → 1`
		`regexp {aX*b} aXb → 1`
		`regexp {aX*b} aXXb → 1`

`+`	Matches 1 or more occurrences of the atom	`regexp {aX+b} ab → 0`
		`regexp {aX+b} aXb → 1`
		`regexp {aX+b} aXXb → 1`

`?`	Matches 0 or 1 occurrences of the atom	`regexp {aX?b} ab → 1`
		`regexp {aX?b} aXb → 1`
		`regexp {aX?b} aXXb → 0`

`{M}`	Matches exactly *M* occurrences	`regexp aX{2}b aXb → 0`
		`regexp aX{2}b aXXb → 1`
		`regexp aX{2}b aXXXb → 0`

`{M,}`	Matches *M* or more occurrences	`regexp aX{2,}b aXb → 0`
		`regexp aX{2,}b aXXb → 1`
		`regexp aX{2,}b aXXXb → 1`

`{M,N}`	Matches *M* to *N* occurrences (inclusive)	`regexp aX{2,4}b aXb → 0`
		`regexp aX{2,4}b aXXXb → 1`
		`regexp aX{2,4}b aXXXXXb → 0`

10.1.6. Alternation and branches

Regular expressions can be combined using the `|` metacharacter to form a RE that will match strings that match **any** of the expressions being combined. Each subexpression is termed an *alternative* or a *branch* of the combined expression. For example, the RE `apple|banana` would match either `apple` or `banana`.

The alternation metacharacter binds at a low precedence so `apple|banana` is equivalent to `(apple)|(banana)` and not `appl(e|b)anana`.

Any of the following would match day of the week.

```
regexp {Sunday|Monday|Tuesday|Wednesday|Thursday|Friday|Saturday} Monday → 1
regexp {(Sun|Mon|Tues|Wednes|Thurs|Fri|Sat)day} Friday                → 1
regexp {(Mon|Wednes|Fri|T(ues|hurs)|S(at|un))day} Tuesday             → 1
```

10.1.7. Constraints

A regular expression *constraint* matches the empty string (i.e. it does not "consume" any characters in the string being matched) **but only when certain conditions are met**. An example of such a condition might be the beginning of a line or word.

10.1.7.1. Anchoring with ^ and $

As we saw above, the regular expression *RE* will match if it matches any substring of *STRING*. If instead we want to check that the *RE* matches *all* of *STRING*, we can *anchor* the RE with the metacharacters `^` and `$`. The former constrains the match to start at the beginning of the string while the latter constrains the *RE* to match the end of the string.

```
regexp {^XY} aXY → 0
regexp {^XY} XYb → 1
regexp {XY$} aXY → 1
regexp {XY$} XYb → 0
```

They may of course be used in combination to force the entire string to match.

```
regexp XY aXYb     → 1
regexp {^XY$} aXYb → 0
regexp {^XY$} XY   → 1
```

The options `-line` and `-lineanchor` alter the semantics on the `^` and `$` anchors (Section 10.1.14).

10.1.7.2. Constraint escapes

Tcl also defines *position based constraints* via the escape sequences shown in Table 10.5.

Table 10.5. Constraint escape sequences

Escape	Description	Example
`\A`	Matches at the beginning of the string similar to `^` but does not change behaviour when newline sensitive matching (Section 10.1.14) is in effect.	`regexp {\AX} "aXb" → 0` `regexp {\AX} "Xab" → 1`
`\Z`	Matches at the end of the string similar to `$` but does not change behaviour when newline sensitive matching (Section 10.1.14) is in effect.	`regexp {X\Z} "aXb" → 0` `regexp {X\Z} "abX" → 1`
`\m`	Matches at the beginning of a word.	`regexp {\mX} "aXb"  → 0` `regexp {\mX} "a Xb" → 1`
`\M`	Matches at the end of a word.	`regexp {X\M} "a Xb" → 0` `regexp {X\M} "aX b" → 1`

`\y`	Matches at the beginning or the end of a word.	`regexp {\yX} "aXb"  → 0`
		`regexp {\yX} "a Xb" → 1`
		`regexp {X\y} "aX b" → 1`

`\Y`	Matches when *not* at the beginning or the end of a word.	`regexp {\YX} "aXb"  → 1`
		`regexp {\YX} "a Xb" → 0`
		`regexp {X\Y} "aX b" → 0`

The word related constraints, `\m`, `\M`, `\y` and `\Y` distinguish word characters in the same manner as the `\w` character escape (Table 10.3).

10.1.7.3. Lookahead constraints

A *lookahead constraint* is based on matching a subexpression **without actually including the matched text as part of the match**. Lookahead comes in two forms:

- *Positive lookaheads* have the form `(?=LOOKAHEAD)` where `LOOKAHEAD` is the RE that should be matched at that point.
- *Negative lookaheads* have the form `(?!LOOKAHEAD)` and are similar except that the `LOOKAHEAD` must **not** be matched.

As an example, here is a RE that matches part numbers that are at most ten characters starting with of one or more uppercase alphabetic characters followed by one or more digits.

```
^(?=.{2,10}$)[A-Z]+[0-9]+$
```

We can break this up into two parts. The first part of the RE is the lookahead

```
(?=.{2,10}$)
```

This ensures the length conditions are met (between 2 and 10 characters in the string) but does not say anything about the expected format. Crucially, it does not consume any characters from the string being matched so that the second part

```
[A-Z]+[0-9]+
```

is matched starting at the same point as the lookahead expression.

We can then do syntactic checks for valid part numbers as

```
set re {^(?=.{2,10}$)[A-Z]+[0-9]+$} → ^(?=.{2,10}$)[A-Z]+[0-9]+$
regexp $re A0                       → 1
regexp $re ABC2345678               → 1
regexp $re 1234567890               → 0 ❶
regexp $re ABC12345678              → 0 ❷
```

❶ Only digits
❷ More than 10 characters

10.1.8. Back references

Consider detection of adjacent repeated words in a document, for example the word *has* in the sentence below. We can construct a RE to detect this.

```
regexp {\mhas\s+has\M} "This sentence has has repeated words." → 1
```

However this is not a general solution given that we do not know a priori which word might be repeated. Instead we have to match words using generic regular expressions. We then need a mechanism that lets us specify the next part of a RE based on a preceding matched subexpression. Back references provide exactly that capability.

A back reference in a RE is specified in the form `\N` where `N` is a number which references a group enclosed by capturing parenthesis (Section 10.1.4). When multiple groups are present, the corresponding "captures" are numbered in the order of the position of their opening parenthesis.

To solve our problem then, the matching RE should

1. begin at the start of a word indicated by the `\m` constraint
2. followed by *any* word matched as `\w+`
3. followed by any amount of whitespace matched as `\s+`
4. followed by the **same string** that was just matched by the above `\w+`
5. followed by the end of word constraint `\M`.

In effect, in step 4 we have to refer back to the word matched in step 2. To do this we enclose the word specifier in capturing parenthesis as `(\w+)` so that the result of its match can be referenced through a back reference. Since this is the only, and therefore the first, capturing parenthesis in the expression, it is referenced as `\1`.

Thus the entire matching expression becomes that shown below:

```
% regexp {\m(\w+)\s+\1\M} "This sentence has has repeated words."
→ 1
% regexp {\m(\w+)\s+\1\M} "This sentence has no repeated words."
→ 0
% regexp {\m(\w+)\s+\1\M} "To be or not to be." ❶
→ 0
```

❶ Repeated but not consecutive

Back references are especially useful when substituting using regular expressions and we will see examples of their use with the `regsub` command (Section 10.2).

10.1.9. Counting number of matches

A regular expression may match multiple times in a string. If the `-all` option is specified, the command will return the number of matches found in the string.

```
regexp -all X+ aXXbXCXXX → 3
```

10.1.10. Retrieving matches

Up to this point, we have only dealt with the simplest form of the `regexp` command — one that tells us whether a given string matches a RE or not. We now look at the various means of having `regexp` actually tell us **what** was matched.

10.1.10.1. Retrieving matched content

Any additional arguments passed to `regexp` are treated as names of variables in which the match results are to be stored. If the RE matches in the example below, `regexp` stores the matched content in the passed variable `xes`. If no match occurs, the variable is unchanged.

```
regexp X+ aXXXc xes → 1
set xes             → XXX
```

If the content of matched subexpressions is of interest, additional variables can be specified. Matches for subexpressions enclosed in capturing parenthesis are successively stored in any specified variables. Non-capturing subexpression matches are not stored.

```
regexp {(X+)(?:Y+)(Z+)} aXXYYZZZb match xes zes → 1
puts "$match, $xes, $zes"                       → XXYYZZZ, XX, ZZZ
```

10.1.10.2. Retrieving matched indices

In some situations, it is more useful to retrieve the string indices of the matches than the actual content itself. Specifying the `-indices` option stores in each specified variable a pair consisting of the start and end indices of the corresponding match.

```
% regexp -indices {(X+)(?:Y+)(Z+)} aXXYYZZZb match xes zes
→ 1
% puts "$match, $xes, $zes"
→ 1 7, 1 2, 5 7
```


When parsing large amounts of text using regular expressions, storing indices, rather than matched content, is often more efficient in time and space.
The original text being parsed is maintained as the “master” copy and the consumer of the parse can use the indices to retrieve substrings as needed.

10.1.10.3. Retrieving matches with -inline

Instead of storing matches in variables, you can have `regexp` return the matches by specifying the `-inline` option.

The return value from `regexp` is a list containing the same values as would have been stored in any variable name arguments if `-inline` was not specified.

```
regexp -inline {(X+)(?:Y+)(Z+)} aXXYYZZZb          → XXYYZZZ XX ZZZ
regexp -inline -indices {(X+)(?:Y+)(Z+)} aXXYYZZZb → {1 7} {1 2} {5 7}
```

If the RE does not match, the command will return an empty list.

```
regexp -inline {(X+)(?:Y+)(Z+)} aYYZZZb → (empty)
```

10.1.10.4. Retrieving all matches

As we saw earlier, the `-all` option can be specified to count the number of matches found. If variables are specified for the command, only the results corresponding to the last match found will be stored in them.

```
regexp -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ match xes zes → 2
puts "$match, $xes, $zes"                               → XYZ, X, Z
```

If you want information for all matches, not just the last one, use the inline version. The result is a flat list containing all matches and submatches.

```
% regexp -inline -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ
→ XXYYZZZ XX ZZZ XYZ X Z
% regexp -inline -indices -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ
→ {1 7} {1 2} {5 7} {9 11} {9 9} {11 11}
```

10.1.11. Option metasyntax

Some `regexp` command options can instead be embedded into the RE by beginning the expression with the metasyntax `(?OPTS)` where *`OPTS`* is a sequence of one or more characters, each corresponding to an option. For example, `i` and `n` corresponds to the options `-nocase` and `-line` respectively, so the two statements

```
regexp -nocase -line {RE} STRING
regexp {(?in)RE} STRING
```

are equivalent. **Embedded options can only appear at the beginning of the RE.**

We will discuss this embedded metasyntax alongside their option equivalents.

10.1.12. Case-independent matching

By default, `regexp` is case sensitive. Passing the `-nocase` option will result in case being ignored. Alternatively, the `(?i)` or `(?c)` metasyntax can be used to specify case-insensitive and case-sensitive matching respectively.

```
regexp xy axyb          → 1
regexp xy aXYb          → 0
regexp -nocase xy aXYb → 1
regexp {(?i)xy} aXYb    → 1
regexp {(?c)xy} aXYb    → 0
```

10.1.13. Matching literal strings

Because of its many options, `regexp` can be useful even for exact matching of literal strings. Suppose we wanted to count the number of occurrences of a literal string.

```
set search_string "XY"               → XY
regexp -all $search_string aXYbXcXYd → 2
```

The above works fine when the `search_string` does not contain any literal characters that might be misinterpreted as metacharacters. But if it does, then we get unexpected results.

```
set search_string "X."               → X.
regexp -all $search_string aX.bXcX.d → 3
```

The problem is with `regexp` treating `.` as a metacharacter when we want to actually treat it as a literal character. One solution is to preprocess the search string to escape any metacharacters with a `\` character. An easier way is to prefix the search expression with `***=` indicating that the rest of the expression is to be treated as a literal string.

```
regexp -all "***=$search_string" aX.bXcX.d → 2
```

The above construct `***=` is not useful when the literal is part of a longer regular expression. In that case, the metacharacters in the literal must be escaped, for example with the `regsub` command (Section 10.2).

```
regsub -all {[][*+?{}()<>|.^$\\]} $literal_string {\\&}
```

The `string map` command (Section 4.10) may also be used for this purpose.

10.1.14. Newline-sensitive matching

By default no special treatment is afforded to newline characters embedded in the string being matched. For some use cases, such as matching lines read from a file in a manner similar to `egrep`, this requires reading the file line by line doing a match on each line.

A more efficient method is to use newline-sensitive matching.

- If the `-lineanchor` option is specified, the metacharacters `^` and `$` are treated as matching the beginning and end of a line respectively. The `\A` and `\Z` constraints are unchanged and continue to match the beginning and end of the entire string.
- If the `-linestop` option is specified, the `.` metacharacter is now treated as matching all characters **except** newlines. Similarly, bracket expressions of the form `[^…]` (ie. matching characters **not** in a set) never match a newline.

The `-line` option combines both the options. Thus we can count the number of lines with extraneous trailing whitespace.

```
% set file_content "First line\nSecond line with trailing space   \nThird line \
    with tab\t"
→ First line
  Second line with trailing space
  Third line with tab
% regexp -all {\s+$} $file_content  ❶
→ 1
% regexp -all -line {\s+$} $file_content ❷
→ 2
```

❶ Only sees trailing tab at end of content
❷ Sees all lines ending in whitespace

The metasyntax sequences `(?w)`, `(?p)` and `(?n)` may be used in place of the command options `-lineanchor`, `-linestop` and `-line` respectively.

So the following would be equivalent to the above example.

```
regexp -all {(?n)\s+$} $file_content → 2
```

10.1.15. Matching at an offset: -start

On occasion, such as incrementally parsing a grammar using regular expressions, you need to begin the match from somewhere other than the start of the string. The `-start` option allows you to do so.

```
regexp -inline -all a+ "aaabacaa"          → aaa a aa
regexp -start 4 -inline -all a+ "aaabacaa" → a aa
```

This feature is commonly useful in conjunction with the `-indices` option where the returned indices are used as the argument to `-start` for the next match attempt.

10.1.16. Controlling greediness

A RE may match a string in multiple ways. Consider the following match

```
% regexp -inline {^(x+)(.*y)$} xxyy
→ xxyy xx yy
```

The RE matches with the first subexpression matching `xx` and the second matching `yy`. The RE could also have matched with the first subexpression matching `x`, and the second `xyy`.

The difference between the two matches is the *greediness* of a match. By default, a quantifier, like `+` above, will match as much as possible. Hence the first subexpression matches the whole sequence of `x` characters.

In some situations, it is desirable to match the *fewest* number of characters possible. The greedy quantifier can be be converted into a non-greedy one by appending a `?`. It will then match the fewest characters required for a successful match.

```
% regexp -inline {^(x+?)(.*y)$} xxyy
→ xxyy x xyy
```

Note the difference from the previous result.

The rules for greediness are detailed in the Tcl reference pages and we will not go into them here other than provide an example where the distinction is useful. Consider extracting content enclosed in an XML tag[1] `<Item>…</Item>`.

We might write an expression as follows

```
% regexp {<Item>(.*)</Item>} "<Item>Item 1</Item>" - content
→ 1
% puts $content
→ Item 1
```

[1] Using regular expressions to parse XML is not recommended in general but is often adequate for quick throwaway scripts.

You will often see `-` or `->` in Tcl regexp commands as placeholders for variable names used for values of no interest.

That however fails when you have multiple tags.

```
% regexp {<Item>(.*)</Item>} "<Item>Item 1</Item><Item>Item 2</Item>" - content
→ 1
% puts $content
→ Item 1</Item><Item>Item 2
```

The problem is again greed, where the `(.*)` expression matches up to the second `</Item>` while we would have wanted it to stop at the first. Appending a `?` to the `*` quantifier to force non-greedy matching gives the desired behaviour.

```
% regexp {<Item>(.*?)</Item>} "<Item>Item 1</Item><Item>Item 2</Item>" - content
→ 1
% puts $content
→ Item 1
```

10.1.17. Comments and expanded syntax

The power of `regexp` is accompanied by related complexity and it can be difficult to discern the purpose of various parts of even a moderately complex RE. Regular expressions in Tcl offer a solution to this problem in the form of an expanded syntax which is enabled by passing the `-expanded` option to the `regexp` command.

Expanded syntax differs from normal RE syntax in the following ways:

- Whitespace in the RE is ignored. You can use spaces and tabs to indent and spread a RE out over multiple lines.
- The `#` character starts a comment and all characters till the end of the line or the expression are ignored.

There are a some exceptions to the above.

- A whitespace or `#` character preceded by a `\` is treated as a significant character and not ignored.
- A whitespace or `#` character within a bracketed expression is significant.
- Whitespace and `#` are illegal within multicharacter symbols. We don't discuss these at all here. See the Tcl reference page for more information.

Here is a previous example for detecting repeated words rewritten in expanded syntax.

```
regexp -inline -all -expanded {
    \m        # Beginning of a word
    (\w+)\s+  # followed by one or more word characters and whitespace
    \1        # then the word that was matched
    \M        # then end of the word (a non-word char, end of string etc.)
} "This sentence has has repeated words."
→ {has has} has
```

Expanded syntax can also be enabled with the `(?x)` metasyntax (Section 10.1.11).

The embedded metasyntax has to be right at the beginning of the regular expression since the expanded syntax begins after the closing parenthesis. Thus there must not be any character, including space or newline, preceding the `(?x)` at the start of the expression.

10.2. Substituting regular expressions: regsub

```
regsub ?options? RE STRING SUBSPEC ?VARNAME?
```

The `regsub` command performs substitutions of substrings matching a RE pattern. The modified string may be returned as the result of the command or stored in a variable.

`RE` is the regular expression, *`STRING`* is the string in which substitutions are to be made and *`SUBSPEC`* is the substitution specification. If *`VARNAME`* is not specified, the command returns the substitution result. If *`VARNAME`* is specified, the result is stored in the variable of that name and the number of substitutions is returned.

The substitution string *`SUBSPEC`* can itself refer to elements of the matched *`RE`* pattern, by using one or more back references of the form `\N` where N is a number between 0 and 9: `\0` will be replaced with the string that matched the entire RE, `\1` with the string that matched the first sub-pattern, and so on. You can also use the character `&` in place of `\0`.

```
% regsub {(\d+) (\d+)} "Example: 100 200" {\0 reversed is \2 \1}
→ Example: 100 200 reversed is 200 100
% regsub {(\d+) (\d+)} "Example: 100 200" {& reversed is \2 \1} var
→ 1
% puts $var
→ Example: 100 200 reversed is 200 100
```

Here `\0` and `&` match `100 200` while the back references `\1` and `\2` refer to the capturing parenthesis content `100` and `200` respectively. **The part of the string that does not match the regular expression is preserved.** Thus the string `Example:` is left untouched in the result.

By default, `regsub` only substitutes the first occurrence of the RE. Pass the `-all` option to substitute all occurrences.

Going back to an example we saw earlier, detection of repeated words in text, we can use `regsub` to fix the errors instead of just detecting them.

```
% regsub -all {\m(\w+)(\s+)\1\M} {
    Words are often repeated when
    when a word appears at the end of a line
    line and is repeated on the next.
} {\1}
→
      Words are often repeated when a word appears at the end of a line and is ...
```

The `regsub` command accepts many, but not all, of the options of the `regexp` command, in particular `-nocase`, `-start`, `-line`, `-linestop`, `-lineanchor` and `-expanded`.

```
% regsub -all {(c)olor} "Colors colors" {\1olour}
→ Colors colours
% regsub -nocase -all {(c)olor} "Colors colors" {\1olour}
→ Colours colours
```

These options have the same effect as for the `regexp` command and we do not further describe them further here.

The following example from RosettaCode[2] illustrates the combined use of `string map`, `regsub` and `subst` to decode URL's. You will often find this combination of commands used in tasks involving decoding operations.

```
proc urlDecode {str} {
    set specialMap {"[" "%5B" "]" "%5D"}
    set seqRE {%([0-9a-fA-F]{2})}
    set replacement {[format "%c" [scan "\1" "%2x"]]}
    set modStr [regsub -all $seqRE [string map $specialMap $str] $replacement]
    return [encoding convertfrom utf-8 [subst -nobackslash -novariable $modStr]]
}
urlDecode "http%3A%2F%2Ffoo%20bar%2F"
→ http://foo bar/
```

Since we have covered all the relevant commands, grokking the code is left as an exercise for the reader.

10.2.1. Computed substitution with regsub

It is often useful to have the substitution performed by `regsub` be the result of a computation rather than some static value or backreference. Consider the problem of URL encoding[3] of a string where a simple but still conforming method is to replace all non-alphanumeric characters with their hexadecimal values preceded by a `%` character. This substitution involves a computation and thus cannot be accomplished by any of the `regsub` features described thus far.

This is where the `-command` option comes into use. In the presence of the `-command` option, the substitution argument to `regsub` is treated as a command to be invoked whose result is then used as the substitution value. This callback command is passed at least one argument, the value of the matched RE. Additional arguments, if present, hold the values of any matched subexpressions in the RE.

The `-command` option is not available in Tcl 8.6 and earlier.

Our motivating example above can be implemented as follows.

[2] https://www.rosettacode.org/wiki/URL_decoding#Tcl
[3] https://en.wikipedia.org/wiki/Percent-encoding

```
% proc enc {ch} {format %%%02X [scan $ch %c]}
% regsub -command -all {[^0-9A-Za-z]} "some-random+string" enc
→ some%2Drandom%2Bstring
```

The RE matches all non-alphanumeric characters. Each such character is passed to the `enc` proc passed as the substitution specification, and replaced with the return value.

The convenience of the command arises from its succint combination of iteration (with the `-all` option), selective matching of the iteration operand, and execution of code of any complexity.

Some further examples that illustrate use with subexpressions.

Ensure the first character after punctuation is upper case:

```
set text "First sentence. second sentence? third sentence."
regsub -command -all {([.!?])\s+(.)} $text {
    apply {{- punc ch} {return "$punc [string toupper $ch]"}}
}
→ First sentence. Second sentence? Third sentence.
```

Convert Markdown headings to HTML:

```
proc md2html {- level text} {
    set h h[string length $level]
    return "<$h>$text</$h>"
}
set mdLine "# First level heading\n## A second level heading"
regsub -all -line -command {^(#+)\s+(.*)$} $mdLine md2html
→ <h1>First level heading</h1>
  <h2>A second level heading</h2>
```

11

Dates and Time

> Until we can manage time, we can manage nothing else.
>
> — Peter F. Drucker

Tcl can do many things, but sadly cannot manage time, only time values. It can however do a fair number of things with those values:

- Tell you what time it is, even in different timezones and calendars
- Convert to multiple display formats in different locales
- Parse date time strings in various formats
- Perform date and time arithmetic

All functions related to these time manipulation features are implemented by the `clock` ensemble command.

11.1. POSIX seconds and the epoch

Most Tcl commands dealing with time work with time values expressed as the number of seconds since the *epoch* — January 1, 1970, 00:00 UTC. These values may be negative as well, representing a time before the epoch. This representation originally comes from the Unix operating system and is now commonly used in other computing environments. We will refer to these values as *POSIX seconds*.

Tcl time computations do not take into account leap seconds. Time since the epoch is calculated assuming exactly 60 seconds in a minute.

11.2. The Julian, Gregorian and alternate calendars

The `clock` command and documentation make reference to three different calendars, Julian, Gregorian and locale-specific alternatives.

The Julian calendar

Historically, the Julian calendar came on the scene first, introduced in 46 BC. It defined a calendar made up of the 12 months that are in common use today with 365 days in a year with a leap year containing an extra day every four years. The consequent average year length of 365.25 days however did not exactly equal the solar year leading to a three day discrepancy over four centuries.

The Gregorian calendar

The Gregorian calendar is the calendar in accepted international use today. Its official introduction occurred on 15 October 1582 with the aim of resolving the discrepancy in the Julian calendar by changing the rules for leap years. This reduced the average year length to 365.2425 days bringing it closer in duration to the solar year. In addition, the Gregorian calendar compensated for the accumulated difference by removing 10 days from the calendar. The first day of the Gregorian calendar, 15 October 1582, followed the last day of the Julian calendar, 4 October 1582. The official Gregorian calendar introduction occurred on 15 October 1582. The *proleptic* Gregorian calendar extends the calendar definition backward over time.

An additional complication when comparing dates across the two calendars, or converting POSIX seconds to a time string, is that countries adopted the Gregorian calendar at different times. Converting POSIX seconds to dates therefore can also entail specification of a locale in which the conversion is to be done. The `GREGORIAN_CHANGE_DATE` entry in the localization database (Section 11.9) contains the date on which the locale changed calendars. The `clock` command uses this when doing the conversion of POSIX seconds to the Gregorian calendar.

Alternative calendars

Some locales, like the Japanese, have other calendars that are still in common use. The Japanese civil calendar is divided into named *eras* based on the reigning Emperor. Years are then numbered within the era and divided into months and days in month in similar fashion to the Gregorian calendar. The `clock` command includes formatting codes that select these alternative calendars.

11.3. Time zones

When dealing with time zones, the `clock` command retrieves the time zone to be used from one of the following sources in order of preference:

- A time zone specified inside the string being parsed
- A time zone specified by command options `-timezone` or `-gmt`
- The `TCL_TZ` and `TZ` environment variables (in that order)
- The local time zone from system settings (Windows only)
- The C runtime library

The time zone strings may take one of several formats:

- Standardized location names begin with a `:`, for example `:America/Argentina/Buenos_Aires`. A full list of location names can be found either under `lib/tcl`*`VERSION`*`/tzdata` or under a system specific directory like `/usr/share/zoneinfo` on Unix systems. The string `:localtime` is a special case that refers to the local time zone as defined by the C runtime library.
- A second form is a string starting with a `+` or `-`, denoting a time zone east or west of Greenwich respectively, followed by a two digit hour offset, a two digit minute offset, and optionally a two digit seconds offset. For example, `+0530` is five and a half hours east of Greenwich and `-080030` is eight hours and thirty seconds west.

- A string conforming to the Posix definition of the `TZ` environment variable, for example `EST+5` for Eastern Standard Time. Note that the semantics of `+` and `-` are opposite from the second form above with `+` indicating a time zone *west* of the Greenwich and `-` denoting east. See the POSIX specification for the full syntax which allows for daylight savings start, end and offsets.
- Strings that do not match any of the above are prefixed with a `:` and attempted to be handled as location names described in the first item above.

We will see examples of time zone use as we discuss each command.

11.4. Retrieving the current time: clock seconds | milliseconds | microseconds

Commands `clock seconds`, `clock milliseconds` and `clock microseconds` return the current time as the number of seconds, milliseconds and microseconds since the epoch.

```
clock seconds      → 1743694529
clock milliseconds → 1743694529406
clock microseconds → 1743694529406529
```

11.5. Interval measurement: clock clicks

While the above commands return time as the number of elapsed time units since the epoch, the `clock clicks` command returns a high-resolution system-dependent value that is not tied to any epoch.

```
clock clicks → 3776835998804
```

The return value from the command cannot be converted to a date and time in any calendar. Rather the difference between two return values can be used for measuring intervals with the highest resolution offered by the platform.

11.6. Formatting time for display: clock format

```
clock format TIMEVAL ?OPTIONS?
```

The `clock format` command formats a time value, given as number of seconds since the epoch, into a string of a specified format that is suitable for display or for passing to other programs.

Without any options, the command will return a string using a default format and locale with the local time zone. The special value `now` can be used to specify the current time.

```
clock format [clock seconds] → Thu Apr 03 21:05:29 IST 2025
clock format now             → Thu Apr 03 21:05:29 IST 2025
```

The `now` value is not supported in Tcl 8.6 and earlier.

11.6.1. Formatting for a different time zone: -timezone, -gmt

By default, the command will format the time using the default time zone (Section 11.3). A different time zone can be specified with the `-timezone` option.

```
clock format 0                              → Thu Jan 01 05:30:00 IST 1970
clock format 0 -timezone :America/New_York → Wed Dec 31 19:00:00 EST 1969
```

The `-gmt` option is an alias for the UTC time zone.

```
clock format 0 -timezone :UTC → Thu Jan 01 00:00:00 UTC 1970
clock format 0 -gmt 1         → Thu Jan 01 00:00:00 GMT 1970
```

11.6.2. Formatting for a locale: -locale

The `clock format` command accepts the `-locale` option to display the time in a format suitable for a specific locale. Permissible option values are `current`, `system` and any locale identifiers accepted by the `msgcat` (Section 9.3.2) command. The value `current` refers to the locale returned by the `mclocale` command while `system` refers to user preferences if available (such as the registry on Windows), and is synonymous with `current` otherwise.

Note that if `-locale` is not specified, it defaults to the `ROOT` locale, not `current` or `system`.

```
msgcat::mclocale es               → es ❶
clock format now                  → Thu Apr 03 21:05:29 IST 2025 ❷
clock format now -locale current → jue abr 03 21:05:29 IST 2025 ❸
clock format now -locale fr       → jeu. avr. 03 21:05:29 IST 2025 ❹
```

❶ Sets locale to Spanish
❷ Still uses `ROOT` locale
❸ Uses `es` locale
❹ Explicit locale

11.6.3. Controlling display format: -format

The `-format` option offers full control over the output. The option value is a format specification consisting of *format groups*, which are sequences beginning with the `%` character. Each format group is replaced with a specific time component like day or hour as shown in Table 11.1. Characters not part of format groups are passed through unchanged.

```
% clock format now -format "The current time is %r."
→ The current time is 09:05:29 pm.
```

The format groups supported by `clock format`, as well as by the `clock scan` command we will describe later, are shown in Table 11.1.

Table 11.1. Format groups for clock

Format group	Description

`%a`, `%A`	Locale dependent day of the week in short and full form respectively.
	`clock format now -format %a → Thu`
	`clock format now -format %A → Thursday`
	`clock format now -format %A -locale de → Donnerstag`

`%b`, `%B`	Locale dependent month short and full form respectively.
	`clock format now -format %b → Apr`
	`clock format now -format %B → April`
	`clock format now -format %B -locale de → April`

`%c`	Localized representation of date and time of day.
	`% clock format now -format %c`
	`→ Thu Apr  3 21:05:29 2025`
	`% clock format now -format %c -locale de`
	`→ 03.04.2025 21:05:29 +0530`

`%C`	Number of the century `clock format now -format %C → 20`
`%d`	Two digit number of the month.
`%D`	Synonym for `%m/%d/%Y` `clock format now -format %D → 04/03/2025`
`%e`	Day of the month as two digits or a single digit with a leading space.

`%Ec`, `%Ex`, `%EX`, `%Ey`, `%EY`	Correspond to `%c`, `%x`, `%X`, `%y` and `%Y` respectively except that the locale's alternative calendar is used. In the example below, the Japanese civil calendar, which is the alternative calendar in the Japanese locale, is contrasted with the default calendar.
	`% clock format 0 -locale ja -format %Y -gmt 1`
	`→ 1970`
	`% clock format 0 -locale ja -format %EY -gmt 1`
	`→ 昭和45`
	`% clock format 0 -locale ja -format %c -gmt 1`
	`→ 1970/01/01  0:00:00 +0000`
	`% clock format 0 -locale ja -format %Ec -gmt 1`
	`→ 昭和45年01月01日（木）00時00分00秒 +0000`
	In the output above, the epoch year 1970 is shown as year 45 in the 昭和, or Shōwa, era.

`%EC`	The locale-dependent era in the locale's alternative calendar `clock format 0 -locale ja -format %EC -gmt 1 → 昭和`

`%EE`	Either string `B.C.E.` or `C.E.`, or their localized versions, depending on whether `%Y` refers to dates before or after Year 1 of the Common Era.
	`clock format 0 -format %EE → C.E.`
	`clock format 0 -format %EE -locale de → n. Chr.`

`%Ej`, `%EJ`	Astronomical Julian Date. This is a floating point number of days since January 1, 4713 BCE noon or midnight respectively. Not in Tcl 8.6.
	`clock format 0 -format %Ej → 2440587.72916667`
	`clock format 0 -format %EJ → 2440588.22916667`

`%Es`	The *TIMEVAL* argument is treated as number of seconds since Jan 1, 1970 local time instead of POSIX seconds. Not in Tcl 8.6.
`%g`, `%G`	A 2-digit and 4-digit year suitable for use with the week based calendar defined in the ISO8601 standard.
`%h`	Same as %b.
`%H`, `%I`	Two digit hour of the day on a 24 and 12 hour clock respectively.
`%j`	A 3-digit day of the year. `clock format 0 -format %j → 001`

`%J`	The Julian day number. This is often useful in calendar calculations.
	`proc julian {secs} {return [clock format $secs -format %J]}`
	`proc days_since {date} {`
	`  set then [clock scan $date -format "%Y/%m/%d"]`
	`  return [expr {[julian [clock seconds]] - [julian $then]}]`
	`}`
	`puts "World War II ended [days_since 1945/09/02] days ago."`
	`→ World War II ended 29068 days ago.`

`%k`, `%l`	One or two digit hour of the day using a 24- and 12-hour clock respectively. Single digit hours are left padded with a space. `clock format 0 -format "(%l)" → ( 5)`

`%m`, `%N`	Number of the month where `%m` always produces a 2-digit value while `%N` left pads single digit months with a space.
	`clock format 0 -format (%m) → (01)`
	`clock format 0 -format (%N) → ( 1)`

`%M`	A 2-digit minute of the hour (00-59).
`%Od`, `%Oe`, `%OH`, `%OI`, `%Ok`, `%Ol`, `%Om`, `%OM`, `%OS`, `%Ou`, `%Ow`, `%Oy`	Correspond respectively to the format groups without the `O` except that they use locale-dependent alternative numerals.

`%p`, `%P`	Outputs a locale-specific `AM`/`PM` (`%p`) or `am`/`pm` (`%P`) indicator. If the locale supports both lower and upper case variations, `%p` and `%P` select the upper and lower case forms respectively.
	`clock format now -format "%H:%M %p" → 21:05 PM`
	`clock format now -format "%H:%M %P" → 21:05 pm`

`%Q`	Reserved for internal use.
`%r`	Locale-dependent time of day using a 12-hour clock. `clock format now -format %r → 09:05:29 pm`
`%R`	Hours and minutes as 24-hour clock. Same as `%H:%M`.

`%s`	Formats the *`TIMEVAL`* argument as a decimal string.
	`% clock format now -format "It is %s seconds since the epoch."`
	`→ It is 1743694529 seconds since the epoch.`

`%S`	The 2-digit second of the minute.
`%t`	Outputs a tab character
`%T`	Time of day. Alias for `%H:%M:%S`.

`%u`, `%w`	The number for the day of the week. `%u` conforms with the ISO8601 standard with days Monday-Sunday numbered 1-7 while `%w` numbers Sunday-Saturday as 0-6.
	`% clock format now -format "Today, %A, is day %u of the week."`
	`→ Today, Thursday, is day 4 of the week.`

`%U`, `%V`, `%W`	The ordinal number of the week in the year. `%U` returns a number in range `00-53` with the first Sunday of the year being the first day of week `01`. `%W` is similar except for week `01` beginning on the first Monday of the year. The preferred grouping `%V`, which conforms to ISO8601 week numbering, returns in the range `01-53`.

`%x`, `%X`	Locale-dependent date and time representation respectively.
	`% clock format now -format "%x %X"`
	`→ 04/03/2025 21:05:29`
	`% clock format now -format "%x %X" -locale be`
	`→  3.04.2025 21.05.29`

`%y` , `%Y`	The 2-digit year of the century and 4-digit calendar year respectively. Note that neither yields the correct value for use with ISO8601 week numbers for which `%g` and `%G` should be used instead.

`%z` , `%Z`	Returns the current time zone in `+/-hhmm` and name format respectively.
	`clock format 0 -format %z -timezone :America/New_York → -0500`
	`clock format 0 -format %Z -timezone :America/New_York → EST`

`%%`	Outputs a single `%` character

`%+`	Same as `%a %b %e %H:%M:%S %Z %Y`.
	`% clock format 0 -format %+`
	`→ Thu Jan  1 05:30:00 IST 1970`

11.7. Parsing dates and times: clock scan

```
clock scan TIMESTRING ?OPTIONS?
```

The `clock scan` command parses a time string returning the equivalent number of seconds since the epoch.

11.7.1. Specifying the parse format: -format

The `-format` option specifies the expected format of the input string.

Although not mandatory, it is strongly recommended that the `-format` be **always** specified. In the absence of this option the command uses heuristics (Section 11.7.6) leading to unexpected results.

The option value takes the form described earlier for `clock format` (Section 11.6.3) except that the format groups define what time components are expected in the **input** *TIMESTRING* argument. The format groups shown in Table 11.1 also apply to `clock scan` so we will not repeat them here but just show some examples.

Parse a full date and time specification:

```
% clock format [clock scan "19900613 003000" -format "%Y%m%d %H%M%S"]
→ Wed Jun 13 00:30:00 IST 1990
```

If the time is not specified, `clock scan` assumes `00:00:00`.

```
% clock format [clock scan "1990-06-13" -format "%Y-%m-%d"]
→ Wed Jun 13 00:00:00 IST 1990
```

If the date is not specified, the current date is assumed unless the `-base` option (Section 11.7.5) is specified. Note the use of the AM/PM designator `%p` in the example below.

```
% clock format [clock scan "12:30am" -format "%I:%M%p"]
→ Thu Apr 03 00:30:00 IST 2025
```

A feature of `clock scan` is that it parses embedded fields in strings.

```
% set t [clock scan "October 27, 2004 - a memorable day in history!" \
           -format "%B %d, %Y - a memorable day in history!"]
→ 1098815400
% clock format $t
→ Wed Oct 27 00:00:00 IST 2004
```

However, the fact that the non-time related characters must exactly match the format string limits its usefulness in parsing log files and such.

11.7.2. Specifying the time zone for parsing: -timezone, -gmt

Just like for `clock format`, the `-timezone` option can be specified to indicate which time zone should be assumed for the string being parsed.

```
clock scan "19900613 003000" -format "%Y%m%d %H%M%S"                → 645217200
clock scan "19900613 003000" -format "%Y%m%d %H%M%S" -timezone :UTC → 645237000
clock scan "19900613 003000" -format "%Y%m%d %H%M%S" -timezone EST  → 645255000
```

The `-gmt` option is a deprecated alias for `-timezone :UTC`.

11.7.3. Parsing localized time strings: -locale

The `clock scan` command accepts the `-locale` option for parsing localized strings. By default, this is the root locale `{}` and not the current locale as returned by `msgcat::mclocale`. The latter can be specified using `-locale current`. On some platforms, the `-locale` option accepts the value `system` for user preferences. On Windows this refers to the user's Control Panel settings. On platforms where user settings are not defined, `system` is synonymous with `current`.

```
% set tstring_fr [clock format 0 -format "%A, %B %d %Y" -locale fr -gmt 1]
→ jeudi, janvier 01 1970
% clock scan $tstring_fr -format "%A, %B %d %Y" -locale fr -gmt 1
→ 0
% clock scan $tstring_fr -format "%A, %B %d %Y" ❶
Ø input string does not match supplied format
```

❶ Error because default locale is not `fr`

11.7.4. Validating time strings: -validate

The `clock scan` command will throw an exception by default if the provided time string is not a valid time value. Since February 29 is not valid except in leap years,

```
% clock scan 2023-02-29 -format %Y-%m-%d
Ø unable to convert input string: invalid day
```

This behaviour can be changed by passing the `-validate` option (defaults to true) with a false boolean value. In this case, the command adjusts the value to bring it within range.

```
% clock format [clock scan 2023-02-29 -format %Y-%m-%d -validate 0] -format \
    %Y-%m-%d
→ 2023-03-01
```

The `-validate` option is not available in Tcl 8.6 and earlier where the behavior is as if the option was passed as false.

11.7.5. Changing the defaults for parsing: -base

When a date is not fully specified, the `clock scan` command uses the *base date* as the default for unspecified components. By default the base date is the current date. In the example below, the year is not specified and will default to the current year.

```
% puts "The current year is [clock format now -format %Y]"
→ The current year is 2025
% clock format [clock scan "01/31" -format "%m/%d"]
→ Fri Jan 31 00:00:00 IST 2025
```

This base date can be changed to use a different date by specifying the `-base` option. The value of the option must be specified as the number of seconds since the epoch. So to use the epoch year as the base date,

```
% clock format [clock scan "01/31" -format "%m/%d" -base 0]
→ Sat Jan 31 00:00:00 IST 1970
```

Or to use year 2000 as the base date,

```
% set secs2000 [clock scan 2000/01/01 -format %Y/%m/%d]
→ 946665000
% clock format [clock scan "01/31" -format "%m/%d" -base $secs2000]
→ Mon Jan 31 00:00:00 IST 2000
```

Note that there is no “base time”; if the time is not specified, it defaults to midnight `00:00:00` in the current locale.

11.7.6. Free form parsing of time strings

When the `-format` option is not specified to the `clock scan` command, it attempts to guess the format of the passed argument. This form is now deprecated because of the ambiguity in interpreting strings and we therefore do not discuss it further.

There is however, one useful form of the free form scan that allows specifying relative time using keywords `now`, `today`, `tomorrow`, `yesterday`, `next`, `last` and `ago`. Some examples:

```
clock format [clock scan now]           → Thu Apr 03 21:05:29 IST 2025
clock format [clock scan tomorrow]      → Fri Apr 04 00:00:00 IST 2025
clock format [clock scan "next week"]   → Thu Apr 10 00:00:00 IST 2025
clock format [clock scan "last month"]  → Mon Mar 03 00:00:00 IST 2025
clock format [clock scan "2 years ago"] → Mon Apr 03 00:00:00 IST 2023
```

The author's recommendation is to avoid surprises by restricting free form scanning to unambiguous simple keywords like `yesterday`.

11.8. Time arithmetic: clock add

```
clock add TIMEVAL ?COUNT UNIT …? ? OPTIONS?
```

Tcl provides for basic time arithmetic operations with the `clock add` command. In its simplest form, it takes *TIMEVAL*, in the form of POSIX seconds, and one or more pairs of arguments that specify the number and unit by which the *TIMEVAL* is to be changed. `TIMEVAL` may also take the value `now` signifying the current time in POSIX seconds.

```
set now [clock seconds]                → 1743694529
clock format $now                      → Thu Apr 03 21:05:29 IST 2025
clock format [clock add $now 1 week]   → Thu Apr 10 21:05:29 IST 2025
clock format [clock add now -2 hours] → Thu Apr 03 19:05:29 IST 2025
```

The unit of time may be one of `years`, `months`, `weeks`, `days`, `weekdays`, `hours`, `minutes` or `seconds` with singular forms of these accepted as well.

```
% clock format [clock add $now 2 years 1 month 1 day]
→ Tue May 04 21:05:29 IST 2027
% clock format [clock add $now 1 day 1 day 1 day] ❶
→ Sun Apr 06 21:05:29 IST 2025
```

❶ Same as `clock add $now 3 days`

The `clock add` command supports the `-timezone` and `-locale` options as described for `clock format` and `clock scan`. The `-timezone` option is important for arithmetic across daylight savings boundaries as we will see below. The `-locale` option determines the date used as the transition from the Julian to the Gregorian calendars which differs in different parts of the world.

11.8.1. Clock computations

There are several complexities in time related computations due to the varying length of time units like months. Here we only provide a summary and refer you to the Tcl command reference for full details and the finer points.

Adding seconds, minutes and hours

Hours and minutes are converted to seconds by multiplying with 3600 and 60 respectively with leap seconds being ignored.

Adding days, weekdays and weeks

Adding days and weeks is done by first converting *TIMEVAL* into a calendar day (**not** date). The days, or weeks multiplied by 7, are then added to the calendar day and then converted back into seconds. Weekdays work similarly to days except that Saturdays and Sundays are skipped.

Accordingly, adding `24 hours` and adding `1 day` do not always have the same effect. Here is an example from the Tcl reference pages that illustrates the difference when the change crosses a Daylight Savings Time boundary.

```
% set t [clock scan {2004-10-30 05:00:00} \
            -format {%Y-%m-%d %H:%M:%S} \
            -timezone :America/New_York]
→ 1099126800
% set tplus1day  [clock add $t 1 day -timezone :America/New_York]
→ 1099216800
% clock format $tplus1day -format %T -timezone :America/New_York
→ 05:00:00
% set tplus24hrs [clock add $t 24 hours -timezone :America/New_York]
→ 1099213200
% clock format $tplus24hrs -format %T -timezone :America/New_York
→ 04:00:00
```

There are additional special cases for daylight savings changes. See the Tcl reference documentation as to how these are handled.

Adding months and years

Adding months and years works similar to adding of days and weeks except that *TIMEVAL* is first converted to the calendar date, **not** day. The months or years are then added to the calendar date as appropriate. If the resulting date is invalid because the month has fewer days, it is set to the last day of the month.

```
% set t [clock scan {2016-05-31} -format %Y-%m-%d]
→ 1464633000
% set tplus1month [clock add $t 1 month] ❶
→ 1467225000
% clock format $tplus1month
→ Thu Jun 30 00:00:00 IST 2016
```

❶ June 31 would be invalid

As for arithmetic involving days and weeks, special cases arise related to daylight savings and calendar changes. We again refer you to the Tcl reference for details.

11.9. Localization

We have seen use of the `-locale` option with various commands to format and parse time values using localized names of months and weeks, different formats and representations and so on. This makes use of Tcl's `msgcat` (Section 9.3.2) facility and supports many locales out of the box. Additional locales may be added defining a set of localized strings as described in the TIP 173[1] specification.

Here we present a small example of extending an existing locale, say `fr`, to display dates using a different separator, `|`, when the format group `%x` is specified. As per TIP 173, the relevant entry is `DATE_FORMAT` and we can set it with the following snippet.

```
namespace eval ::tcl::clock {
    ::msgcat::mcset fr_xx DATE_FORMAT "%e|%B|%Y"
}
→ %e|%B|%Y
```

And voilà!

```
clock format 0 -format %x -locale fr_xx →  1|janvier|1970
```

The above snippet would normally be placed in the `fr_xx.msg` file in a directory that is loaded by the `msgcat` package but could also be executed independently. See Section 9.3.2 for more information.

11.10. Time representation standards

ISO 8601 and RFC 2822 are two international standards that define time representation and formats. Tcllib[2] includes two packages that deal with these formats:

- The `clock::iso8601` package implements formats defined in ISO 8601.
- The `clock::rfc2822` package implements the formats defined in RFC 2822.

Since these are not part of the core Tcl functionality, we do not describe them here. A tutorial for the same is available from the book website[3].

[1] http://www.tcl-lang.org/cgi-bin/tct/tip/173
[2] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md
[3] https://www.magicsplat.com/ttpl/index.html

12

Files and File Systems

> We have persistent objects, they're called files.
>
> — Ken Thompson

Operations on files and file systems are part of almost every application. This chapter covers

- Parsing and construction of file paths
- Management of file metadata such as file attributes
- File system operations related to directories and volumes

File I/O and the ZipFS file system are postponed to Chapter 13 and Chapter 25.

12.1. File paths

The `file` command ensemble hosts most of the functions related to paths and file systems.

12.1.1. Path syntax

On Unix and macOS, file paths may contain any character other than a `/` which is used as the file path component separator. Path components `.` and `..` are special and interpreted as the current directory and its parent respectively. Consecutive `/` characters are treated as a single `/` and trailing ones are ignored.

Windows permits both `/` and `\` as separators and interprets `.` and `..` as above. Paths may start with an drive letter or a UNC path of the form `\\COMPUTERNAME\SHARENAME`.

When using the `\` character as a file path separator in a literal path string, remember that Tcl also treats it as a special character. It therefore needs to be doubled or protected inside braces.

Tcl hides platform differences to the extent possible by two means:

- A generic Unix-like syntax that uses `/` as the file path separator, and
- Commands for parsing and constructing file paths from individual path components.

Nevertheless, applications should be aware of differences in file systems such as supported characters and path length limits.

12.1.2. Absolute and relative paths: file pathtype

```
file pathtype PATH
```

A path may absolute, relative to the current directory or relative to a volume. The `file pathtype` command can be used to make the determination.

The command works purely on a syntactic basis so the specified path does not have to be that of an existing file. It returns `absolute`, `relative` or `volumerelative` as appropriate.

```
file pathtype c:/foo/bar                    → absolute
file pathtype {\\RemoteSystem\C_Drive\foo} → absolute
file pathtype foo/bar                       → relative
file pathtype ../foo/bar                    → relative
file pathtype {c:foo\\bar}                  → volumerelative
file pathtype /foo/bar                      → volumerelative ❶
```

❶ Note that on Unix, this would return the value `absolute`.

The last two examples highlight that on Windows systems, where file paths can have a drive component, `volumerelative` may mean the path is either relative to the current working directory on a specifed volume *or* a specific file on the current working volume.

12.1.3. Home directory and tilde substitution

```
file home ?USER?
file tildeexpand PATH
```

It is a common convention in Unix shells to treat paths starting with a tilde, `~`, as relative to the home directory of a user. The commands `file home` and `file tildeexpand` provide similar function.

The `file home` command returns the home directory for the user *USER* or for the current user if no argument is passed.

```
file home → C:/Users/apnad/Documents
```

The `file tildeexpand` command checks if its *PATH* argument begins with a tilde. If the tilde is immediately followed by a path separator, it is replaced by the value of the `HOME` environment variable. Otherwise, all characters between the tilde and the next path separator are treated as the name of a user on the system and the path component is replaced with that user's home directory. The command returns the result after replacement.

```
file tildeexpand ~/some/path      → C:/Users/apnad/Documents/some/path
file tildeexpand ~ashok/some/path → C:/Users/apnad/some/path
```

The `~USER` form does not look at the specified user's `HOME` environment variable. Thus the two forms `~` and `~USER` may not give the same results even when `USER` is the same as the one owning the current process.

The `file home` and `file tildeexpand` commands are not available in Tcl 8.6 and earlier. Instead all commands that accept paths do tilde substitution in the manner described for the `file tildeexpand` command. This behavior has been removed[1] from Tcl 9.

12.1.4. Parsing paths: file dirname | extension | rootname | split | tail

```
file dirname PATH
file extension PATH
file rootname PATH
file split PATH
file tail PATH
```

The *PATH* argument may be either absolute or relative.

The complementary commands `file dirname` and `file tail` return the directory component of the path and the name of the file respectively.

```
file dirname /dir/subdir/file.ext → /dir/subdir
file tail /dir/subdir/file.ext    → file.ext
file dirname foo                  → .
file tail foo                     → foo
file tail foo/                    → foo ❶
```

❶ Note trailing separators are completely ignored.

The `file extension` command returns the extension of a path or an empty string if the path does not have an extension. Conversely, `file rootname` returns the entire path except the extension.

```
file extension /dir/subdir/file.ext → .ext
file extension /dir/subdir/file     → (empty)
file rootname /dir/subdir/file.ext  → /dir/subdir/file
```

The `file split` command returns the components of a path as a list.

```
% file split c:/dir/filename
→ c:/ dir filename
% file split {\\RemoteSystem\ShareName\dir\filename}
→ //RemoteSystem/ShareName dir filename
% file split dir/filename
→ dir filename
```

[1] See https://wiki.tcl-lang.org/page/Tilde+Substitution and https://core.tcl-lang.org/tips/doc/trunk/tip/602.md.

12.1.5. Constructing paths: file join

```
file join PATH ?PATH …?
```

Just as for parsing paths, Tcl provides a command, `file join`, for path construction in a platform-independent manner. *`PATH`* arguments that are relative paths are joined to the path being constructed using a path separator. If a *`PATH`* is an absolute path, the path constructed so far is discarded and the argument becomes the initial value of the constructed path for the remaining arguments.

```
file join dir subdir file.ext          → dir/subdir/file.ext
file join dir/sub1 sub2\\sub3 file.ext → dir/sub1/sub2/sub3/file.ext ❶
file join dir /subdir file.ext         → /subdir/file.ext ❷
```

❶ Note `\` replaced with the Tcl canonical separator `/`.
❷ Absolute path arguments result in previous arguments being ignored.

The `file join` command is often used with a single argument to replace the native platform path separator with Tcl's canonical `/` separator. For example, on Windows

```
file join {c:\windows\system32} → c:/windows/system32
```

12.1.6. Path normalization: file normalize

```
file normalize PATH
```

The `file normalize` command converts a path to an absolute path. *`PATH`* may be relative, volume relative or absolute. In the first two cases, it is converted to an absolute path. In addition, for all three cases the command takes the following actions to generate a unique string to identify the file.

- Removes all `.` and `..` occurences, adjusting other path components as appropriate.
- Replaces all links in the path by their targets with the exception that the last component is not replaced even if a link. This exception is by design in case the application wants to manipulate the link itself and not the link target.
- (Windows only) Replaces any path components that use the 8.3 short name format by the long form name. Moreover, if the path component actually exists, the exact case-sensitive version of the name is used.
- In Tcl 8.6 and earlier, the command also does leading tilde substitution (Section 12.1.3).

The way `file normalize` handles links may not be suitable for some purposes. First, it replaces any links in the path with the target. Second it does *not* replace a link if it is the last component in the path.

If you do want the last component in the file path to also be replaced if it is a link, you can use the following trick. Append a dummy non-existent file name, such as `...`, and normalize the resulting path. Then use `file dirname` to retrieve the original path in normalized form. For example, if `$path` is the path to be normalized,

```
file dirname [file normalize [file join $path ...]]
```

The `fileutil` module of Tcllib[2] includes the `fullnormalize` command that utilizes this trick. The module also contains the `lexnormalize` command which performs normalization purely based on the syntactic structure of its argument with no special consideration for links and without converting 8.3 names to their long form.

The following examples illustrate `file normalize` on Windows systems.

```
% file normalize a/b ❶
→ C:/temp/a/b
% file normalize c:/temp/foo/.././bar ❷
→ C:/temp/bar
% file normalize c:/WINDOWS/system32 ❸
→ C:/Windows/System32
% file normalize c:/WINDOWSX/system32 ❹
→ C:/WINDOWSX/system32
% file normalize AVERYL~1 ❺
→ C:/temp/A very long file name
```

❶ Convert relative to absolute
❷ Removal of `.` and `..`
❸ Fix character case for path components that exist
❹ Path components that do not exist keep existing character case
❺ Convert file short name to long name

12.1.7. Converting paths to native form: file nativename

```
file nativename PATH
```

When passing a file path to an external program, for example the command shell on Windows, the path may need to be converted to the native form for that platform. The `file nativename` command does this conversion. For example, on Windows,

```
file nativename /dir/subdir/file.ext → \dir\subdir\file.ext
```

[2] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

12.2. File properties and metadata

Files have properties and other metadata associated with them. Tcl supports several that are common across platforms and file systems.

12.2.1. File size: file size

```
file size PATH
```

The `file size` command returns the number of bytes in the specified file.

```
file size [info nameofexecutable] → 194048
```

In case of links, the command returns the size of the link target, not the link itself.

12.2.2. File timestamps: file atime|mtime

```
file atime PATH ?TIMESTAMP?
file mtime PATH ?TIMESTAMP?
```

The `file atime` and `file mtime` commands get or set file access and modification times respectively. The commands return time values in POSIX seconds (Section 11.1).

```
clock format [file atime [info nameofexecutable]] → Thu Apr 03 21:05:19 IST 2025
clock format [file mtime [info nameofexecutable]] → Fri Jan 03 12:10:54 IST 2025
```

Additionally, if *TIMESTAMP* is specified, it must also be in POSIX seconds and the corresponding timestamp is set to this value.

```
% clock format [file atime [info nameofexecutable] [clock seconds]]
→ Thu Apr 03 21:05:29 IST 2025
```

If *PATH* is a link, the commands operate on the link target, not the link itself.

Not all file systems maintain access and/or modification times or permit them to be set. In such cases the commands will raise an error.

12.2.3. File information: file stat|lstat

```
file stat PATH VAR
file lstat PATH VAR
```

The commands `file stat` and `file lstat` are a direct interface to the `stat` and `lstat` system calls. The two calls only differ when *PATH* refers to a symbolic link. In that case, `file`

`stat` returns information about the file that is the target of the link whereas `file lstat` returns information about the link itself.

Both commands store the result of the system call in an array of name *VAR* in the caller's context. The elements of the array are shown in Table 12.1.

Table 12.1. File stat array elements

Element	Description
`atime`	The last access time of the file in seconds since January 1, 1970.
`ctime`	The creation time of the file in seconds since January 1, 1970.
`dev`	The device id of the device on which the file resides.
`gid`	Group id of the file owner.
`ino`	The inode number of the file.
`mode`	The mode bits from the directory entry for the file.
`mtime`	The last modification time of the file in seconds since the epoch.
`nlink`	The number of hard links to the file.
`size`	The number of bytes stored in the file.
`type`	The type of the file.
`uid`	User id of the file owner.

```
% file stat [info nameofexecutable] stat
% parray stat
→ stat(atime) = 1743694529
  stat(ctime) = 1735886454
  stat(dev)   = 2
...Additional lines omitted...
```

These commands are very much reflective of Unix file systems and many elements do not make sense for all platforms. Their use should therefore be avoided in portable code.

12.2.4. Access checks: file exists | readable | writable | executable | owned

```
file exists PATH
file executable PATH
file owned PATH
file readable PATH
file writable PATH
```

The above commands determine the accessability of a file. All these return `1` if the file identified by the path exists **and** is accessible for the specified mode, and `0` otherwise.

The `file exists` command simply checks if the file or directory exists.

```
file exists c:/windows → 1
```

The `file readable`, `file writable` and `file executable` commands check if the file can be read, written or executed by the current real (not effective) user id.

```
file readable c:/windows/system32/cmd.exe   → 1
file readable nosuchfile                    → 0
file writable c:/windows/system32/cmd.exe   → 0
file executable c:/windows/system32/cmd.exe → 1
```

The commands also apply to directories, indicating whether the contents can be listed, files created and the directory itself traversed.

```
file readable c:/windows   → 1
file writable c:/windows   → 0
file executable c:/windows → 1
```

Finally, the `file owned` command indicates if the file is owned by the current user.

```
file owned [info nameofexecutable] → 1
```

There are several caveats that apply to the use of all these access check commands. For example, a return value of `1` from `file readable` does not always allow the file to be actually read for several reasons. It may be locked by another process, protected by Windows integrity levels etc. Thus, in the author's opinion it is advisable to just attempt the desired operation, like opening the file for read access, and then handle any error exceptions.

12.2.5. File types: file isdirectory | isfile | type

```
file isdirectory PATH
file isfile PATH
file type PATH
```

The `file type` command returns one of `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket` indicating the type of the file.

```
file type c:/windows/system32/cmd.exe → file
file type c:/windows/system32         → directory
file type CON                         → characterSpecial ❶
```

❶ The special Windows built-in console interface

For the common case where we need to know if a file is a regular file or a directory, the `file isfile` and `file isdirectory` commands offer a convenient means.

```
file isfile nosuchfile                  → 0
file isfile c:/windows/system32/cmd.exe → 1
file isfile $env(WINDIR)                → 0
file isdirectory $env(WINDIR)           → 1
```

12.2.6. File attributes: file attributes

```
file attributes PATH
file attributes PATH ATTRIBUTE
file attributes PATH ?ATTRIBUTE VALUE …?
```

File systems may store attributes associated with a file. For example, Windows stores a short 8.3 version of a file name along with its real name. The `file attributes` command retrieves, and in some cases stores, these file system specific attributes.

The *`PATH`* argument specifies the file whose attributes are to be accessed. If it refers to a link, the attributes are those of the link's target, not the link itself.

The first form returns all attributes for the specified file. The second form returns the value of the specified attribute. The third form sets the values of one or more attributes. Note that not all attributes can be set. The permitted values of *`ATTRIBUTE`* are platform-dependent and shown in the tables below.

Unix file attributes

The file attributes on Unix platforms are shown in Table 12.2.

Table 12.2. Unix file attributes

Attribute	Description
`-group`	The group attribute. Either the group name or id can be passed when setting. The return value is always the name.
`-owner`	Name of user owning the file. Either the name or id can be passed when setting. The return value is always the name.
`-permissions`	The numeric code as accepted by the `chmod` system call. When being set, the command will accept a symbolic form as well. For example, `u+rw` or `rwxr—r--` . See the reference documentation for details.
`-readonly`	The readonly attribute for a file on Unix systems that support the `uchg` flag for the `chflags` system call.

Windows file attributes

File attributes for Windows systems are shown in Table 12.3.

Table 12.3. Windows file attributes

Attribute	Description
`-archive`	Retrieves or sets the value of the *archive* file attribute.
`-hidden`	Retrieves or sets the value of the *hidden* file attribute.
`-longname`	Returns the long name for the path, converting each component of the path to its long name. Read-only.
`-readonly`	Retrieves or sets the value of the *readonly* file attribute.
`-shortname`	Returns the 8.3 format name for the path, converting each path component to its short name. Read-only.
`-system`	Retrieves or sets the value of the *system* file attribute.

The following shows the equivalence between the short and long names for a file.

```
set long_path "c:/temp/A long name.long extension"     → c:/temp/A long name.long
 extension
close [open $long_path w]                              → (empty)
file exists $long_path                                 → 1
set short_path [file attributes $long_path -shortname] → C:/TEMP/ALONGN~1.LON
file attributes $short_path -longname                  → C:/TEMP/A long name.long
 extension
file normalize $short_path                             → C:/TEMP/A long name.long
 extension ❶
file exists $long_path                                 → 1
file delete $short_path                                → (empty)
file exists $long_path                                 → 0
```

❶ Remember that in addition to converting paths to absolute paths, `file normalize` also maps paths to their long name format.

Retrieving the short name of a file is often useful when executing external programs with `exec` (Section 20.1). Short names never contain spaces, obviating the need for escaping space characters in any file name passed to the external program.

macOS file attributes

File attributes on macOS systems are shown in Table 12.4.

Table 12.4. macOS file attributes

Attribute	Description
`-creator`	The Finder creator type of the file.
`-hidden`	Retrieves or sets the value of the *hidden* file attribute.
`-readonly`	Retrieves or sets the value of the *readonly* file attribute.
`-rsrclength`	Length of the resource fork. When setting, only `0` is accepted and results in the resource fork being stripped from the file.

12.3. File system operations

12.3.1. File system information: file volumes|system|separator

```
file separator ?PATH?
file system PATH
file volumes
```

The `file separator` command returns the separator used by the file system containing the specified path or the native file system if no path is passed.

```
file separator [zipfs root] → / ❶
file separator              → \
```

❶ Virtual file system mount (Section 25.2.2).

The `file system` command returns the type of a file system. The argument `PATH` can be any path on the filesystem of interest. The returned list contains one or two elements, the first identifying the file system and the second, if present, the specific type.

```
file system c:/       → native NTFS
file system //zipfs:/ → zipfs zip
```

The `file volumes` command returns the list of volumes mounted on the system. On Windows, the command returns the list of drives, remote shares and mounted VFS volumes. On Unix, the returned list includes `/` and VFS volumes.

```
file volumes → //zipfs:/ C:/ D:/
```

12.3.2. Creating directories: file mkdir

```
file mkdir ?DIR …?
```

The `file mkdir` command creates one or more directories. For each argument specified, the command will create a directory with that path including any intermediate directories. If a path already exists, no action is taken if it is a directory and an error raised otherwise. The arguments are processed in order and in case of errors, processing of further arguments is aborted but any previous directories that have already been created are not removed.

```
file exists /tmp/dirA      → 0
file mkdir /tmp/dirA/dirB  → (empty) ❶
file exists /tmp/dirA/dirB → 1
```

❶ Intermediate directory will also be created.

12.3.3. Removing files and directories: file delete

```
file delete ?-force? ?--? ?PATH …?
```

The `file delete` command deletes files and directories. Each *`PATH`* argument may refer to a file or a directory. For arguments that are symbolic links, the link itself is removed and not its target. If the argument specifies a file or directory that does not exist, it is ignored without raising an error.

The `-force` option affects two failure modes. Normally if the path corresponds to a non-empty directory or is the working directory of the current process, the command will raise an error. If `-force` is specified, the command will delete non-empty directories along with their content and also change the working directory, if required, so as to allow the directory deletion to proceed.

```
cd /tmp/dirA                  → (empty)
file delete /tmp/dirA         Ø error deleting "/tmp/dirA": permission denied ❶
pwd                           → C:/tmp/dirA
file exists /tmp/dirA/dirB    → 1
file delete -force /tmp/dirA → (empty)
pwd                           → C:/tmp ❷
file exists /tmp/dirA/dirB    → 0
```

❶ Will fail for two reasons - current directory and not empty.
❷ Notice current directory changed.

The optional `--` argument indicates the end of options causing all remaining arguments to be treated as paths. In particular, if `-force` follows the `--`, it will be treated as a path argument and not as an option.

12.3.4. Copying and renaming: file copy|rename

```
file copy ?-force? ?--? FROMPATH ?FROMPATH …?  TOPATH
file rename ?-force? ?--? FROMPATH ?FROMPATH …?  TOPATH
```

The `file copy` and `file rename` commands are similar to each other in their behaviour so we describe them together. Both commands conceptually (not necessarily how they are implemented) make a copy of existing files or directories but `file rename` also deletes the original source after making the copy.

The behaviour of these commands is slightly involved due to different variations depending on whether files or directories are being copied, whether the destination path *`TOPATH`* already exists, the number of arguments supplied and so on.

In this description, “copying” a directory also involves recursively copying all files and subdirectories contained within it.

If exactly one *`FROMPATH`* argument is specified, it may be either a file or a directory. The `file copy` command behaves as follows:

- If *TOPATH* does not exist, a copy of *FROMPATH*, whether a file or a directory, is stored as *TOPATH*.
- If *TOPATH* exists and is a **directory**, a copy of *FROMPATH*, again irrespective of whether it is a file or directory, is made and placed **under** *TOPATH*.
- If *TOPATH* exists and is **not a directory**, it is overwritten with a copy of *FROMPATH* if the latter is also not a directory **and** the `-force` option is specified. Otherwise (if *FROMPATH* is a directory or `-force` is not specified) an error is raised.

When the source file is a symbolic link within the same file system as the destination, the link itself is copied and not the link target.

The following examples illustrate the above scenarios. We use the `glob` (Section 12.3.5) command to list the contents of directories.

```
% file copy /temp/fromDir /temp/newDir ❶
% glob /temp/newDir/*
→ C:/temp/newDir/fileA.txt C:/temp/newDir/subDir
% file copy /temp/fromDir /temp/toDir ❷
% glob /temp/toDir/*
→ C:/temp/toDir/fromDir
% file copy /temp/fromDir/fileA.txt /temp/newDir ❸
Ø error copying "/temp/fromDir/fileA.txt" to "/temp/newDir/fileA.txt": file exists
% file copy -force -- /temp/fromDir/fileA.txt /temp/newDir ❹
% file copy -force -- /temp/fromDir/subDir /temp/newDir  ❺
Ø error copying "/temp/fromDir/subDir" to "/temp/newDir/subDir": file exists
```

❶ Creates a recursive copy of `fromDir` **as** `newDir`.
❷ Creates a recursive copy of `fromDir` **under** `toDir` as `toDir` already exists.
❸ Fails because file exists.
❹ Option `-force` forces overwrite of existing file.
❺ Option `-force` will **not** overwrite an existing directory if it is not empty.

Irrespective of whether the `-force` option is specified or not, the command will never overwrite a directory that is not empty (as illustrated above), or overwrite a file with a directory or vice versa.

The above description applies when there is exactly one *FROMPATH* argument. If more than one *FROMPATH* argument is specified, *TOPATH* **must be an existing directory** and `file copy` behaves as the second case above, placing a copy of each *FROMPATH* argument, whether a file or a directory, under the *TOPATH* directory.

```
% file copy /temp/fromDir/subDir/fileB.txt /temp/toDir /temp/newDir
% glob /temp/newDir/*
→ C:/temp/newDir/fileA.txt C:/temp/newDir/fileB.txt C:/temp/newDir/subDir
   ↳ C:/temp/newDir/toDir
% file copy /temp/fromDir/subDir/fileB.txt /temp/toDir /temp/newDir2 ❶
Ø error copying: target "/temp/newDir2" is not a directory
```

❶ Fails because /temp/newDir2 is not an existing directory.

The `file rename` command behaves similarly except that for all successful copies, `file rename` will delete the original file. As an implementation detail, when both the source and destination are on the same file system, this “copy and delete” operation may in fact be a single “move” or “rename” operation.

12.3.5. Enumerating files: glob

```
glob ?OPTIONS? ?--? ?GLOBPAT …?
```

The `glob` command returns a list of all files matching any of one or more patterns. The returned list contains matching files in an unspecified order. Each *`GLOBPAT`* is a pattern as described for the `string match` command (Section 4.24) with two additional features. A pair of braces containing strings separated by commas can be used to enclose alternatives in a pattern. Secondly, a pattern ending in a `/` will only match directories, not ordinary files. The special characters are shown in Table 12.5.

Table 12.5. Glob patterns

Character	Description
`*`	Matches any number of characters except directory separators.
`?`	Matches any character except a directory separator.
`[…]`	Matches one occurrence of any character between the brackets except directory separators. A range of characters can also be specified. For example, `[a-z]` will match any lower-case letter.
`{`*`STRING`*`?,…?}`	Matches any of the *`STRING`* character sequences separated by commas within the braces except directory separators.
`\`	The backslash escapes the following character such as `*` or `?` so that it is treated as an ordinary character. This allows you to write patterns that match literal glob-sensitive characters, which would otherwise be treated specially.

To illustrate the use of `glob` and its various options, let us first create a directory structure.

```
file mkdir C:/tmp/tcl-book
close [open C:/tmp/tcl-book/foo.txt w] ❶
close [open C:/tmp/tcl-book/fubar.doc w]
close [open C:/tmp/tcl-book/foohidden w] ❷
file attributes C:/tmp/tcl-book/foohidden -hidden 1
file mkdir C:/tmp/tcl-book/foodir
close [open C:/tmp/tcl-book/foodir/foo.txt w]
file mkdir C:/tmp/tcl-book/f{}dir    ❸
close [open C:/tmp/tcl-book/f{}dir/bar.txt w]
```

❶ This creates an empty file using `open` and `close` (Section 13.1).
❷ A hidden file
❸ Directory name with special characters

The following examples illustrate basic `glob` usage.

```
% glob C:/t*/*book/f* ❶
→ C:/TEMP/book/files.adocgen C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/...
% glob C:/tmp/tcl-book/*.txt C:/tmp/tcl-book/foodir/* ❷
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir/foo.txt
% glob C:/tmp/tcl-book/f*/ ❸
→ C:/tmp/tcl-book/foodir/ C:/tmp/tcl-book/f{}dir/
% glob C:/tmp/tcl-book/f{oodi,uba}r ❹
→ C:/tmp/tcl-book/foodir
```

❶ Wild cards can appear in any path component
❷ Multiple patterns
❸ Trailing `/` will only match directories and returned values will also have `/` appended
❹ Example of alternation

In Tcl 8, the command would raise an error if the list of matching files was empty. This is no longer true in Tcl 9. In Tcl 8, the `-nocomplain` option can be passed to have it return an empty list instead. In Tcl 9 this option has no effect.

```
glob nosuchfil*             → (empty)
glob -nocomplain nosuchfil* → (empty)
```

12.3.5.1. Matching based on type: -type option

In addition to matching files based on file name patterns, `glob` can also further qualify matches based on the type of the file and access attributes through the `-type` option. The option value is a list of type and permissions specifiers. These fall into two categories where `glob` will return a file name if it matches **any** specifier from the first category and **all** specifiers from the second category.

The specifiers in the first category are shown in Table 12.6. Returned files will match one of these specifiers that are included in the value for `-type`.

Table 12.6. Glob category 1 type specifiers

Specifier	Description
`b`	Must be a block-special file.
`c`	Must be a character-special file.
`d`	Must be a directory.
`f`	Must be an ordinary file.
`l`	Must be a symbolic link.
`p`	Must be a named pipe.
`s`	Must be a socket.

The specifiers in the second category are shown in Table 12.7. Returned files will match **all** these specifiers that are included in the value for `-type`.

Table 12.7. Glob category 2 type specifiers

Specifier	Description
`r`	File has read permission.
`w`	File has write permission.
`x`	File has execute permission.
`readonly`	File has the read-only attribute.
`hidden`	File has the hidden attribute. By default, `glob` will not include hidden files. With this specifier, `glob` includes **only** hidden files.
`XXXX`	(Mac OS only) File has the 4-character type, e.g. `TEXT`
`{macintosh type` *`XXXX`*`}`	(Mac OS only) File has the 4-character type, e.g. `TEXT`
`{macintosh creator` *`XXXX`*`}`	(Mac OS only) File has the creator *`XXXX`*.

Some examples of filtering based on the types:

```
% glob -type {f d} C:/tmp/tcl-book/fo* ❶
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir
% glob -type d C:/tmp/tcl-book/fo*    ❷
→ C:/tmp/tcl-book/foodir
% glob -type {f hidden} C:/tmp/tcl-book/fo*  ❸
→ C:/tmp/tcl-book/foohidden
```

❶ Lists both files (`f`) and directories (`d`).
❷ Lists only directories.
❸ Lists only ordinary files that are hidden.

12.3.5.2. Changing glob locations: -directory, -path

By default, the `glob` command uses the current directory as the starting context for file matching. For example, `glob *` will return the files in the current directory. The `-directory` and `-path` options allow this context to be changed.

The two differ in that the `-directory` option value is the complete path to a directory while the `-path` option specifies any prefix, even a partial path.

The following commands are equivalent.

```
% glob c:/tmp/tcl-book/*
→ c:/tmp/tcl-book/foo.txt c:/tmp/tcl-book/foodir c:/tmp/tcl-book/fubar.doc c:/t...
% glob -directory c:/tmp/tcl-book *
→ c:/tmp/tcl-book/foo.txt c:/tmp/tcl-book/foodir c:/tmp/tcl-book/fubar.doc c:/t...
```

The convenience of these options arises when the path contains characters that treated as special characters by `glob`. For example, the directory `f{}dir` will get interpreted as `fdir` when passed directly as the `glob` pattern without escaping as there are no elements listed

between the braces (see Table 12.5). When passed via the `-directory` option, the braces are no longer interpreted as glob patterns.

```
% glob c:/tmp/tcl-book/f{}dir/* ❶
% glob c:/tmp/tcl-book/f\\{\\}dir/* ❷
→ c:/tmp/tcl-book/f{}dir/bar.txt
% glob -directory c:/tmp/tcl-book/f{}dir * ❸
→ c:/tmp/tcl-book/f{}dir/bar.txt
% glob -directory c:/tmp/tcl-book *.txt *.doc ❹
→ c:/tmp/tcl-book/foo.txt c:/tmp/tcl-book/fubar.doc
```

❶ Does not match anything as `{}` treated as subpattern list.
❷ Escaped with double `\`, one for Tcl, one for glob pattern.
❸ No escapes needed if the directory component is passed via `-directory`.
❹ If multiple patterns are specified, the directory applies to all.

The `-path` option has a similar effect except that it specifies **any** prefix, not just a directory component. The difference is illustrated by the following:

```
glob -directory c:/tmp/tcl-book/f{}dir/b * → (empty)
glob -path c:/tmp/tcl-book/f{}dir/b *      → c:/tmp/tcl-book/f{}dir/bar.txt
```

In the first case, `glob` looks for a *directory* called `c:/tmp/tcl-book/f{}dir/b` which is not found. In the second case, `glob` uses the passed option value as a *prefix* to be matched.

12.3.5.3. Stripping path names: -tails

The above examples returned full paths of the matching files. In many cases, only the name of the file is desired and not the full path. The `-tails` option provides an easier alternative to iterating over the returned list invoking the `file tail` command for each file. Note that the `-tails` option requires either `-directory` or `-path` to also be specified. Here is an example showing the effect of the option.

```
% glob -directory C:/tmp/tcl-book *
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-book/fubar.doc C:/t...
% glob -directory C:/tmp/tcl-book -tails *
→ foo.txt foodir fubar.doc f{}dir
% glob -path C:/tmp/tcl-book/f -tails *
→ foo.txt foodir fubar.doc f{}dir
```

12.3.5.4. Combining path component patterns: -join

Sometimes the various path components to be used in the pattern match are supplied as separate arguments. We can use the `file join` (Section 12.1.5) command to combine these before passing to `glob`. Alternatively, we can use the `-join` option to `glob` which indicates that all patterns are to be combined as path components.

```
% glob -join C:/tmp tcl* f*
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-book/fubar.doc C:/t...
```

The `glob` command does not recurse into directories. A command such as

```
glob /*/*/*
```

will return file names exactly three levels deep from the root. Writing a recursive version using `glob` is trickier than you might think at first glance having to take into account links, circular references and so on. The Tcllib[3] `fileutil` package includes commands that can be used for the purpose.

12.3.5.5. Special considerations for glob

When using `glob` there are a few considerations to be taken into account because of platform differences and some debatable quirks in the command behaviour. These are listed in this section. **It is important to be aware of these to avoid unexpected results.**

12.3.5.5.1. Case sensitivity

The case-sensitivity of `glob` matching depends on the underlying file system. On Windows for example, the pattern `foo*` will match file `FOOBAR` as well whereas on Unix it will not.

12.3.5.5.2. Short names on Windows

For file names that do not fit the 8.3 filename format, Windows creates a corresponding 8.3 format short name. The `glob` command does *not* pay any heed to these short names when matching special characters. It will however match if the exact file name is specified. For example,

```
% set long_path "/tmp/tcl-book/a long directory name"
→ /tmp/tcl-book/a long directory name
% file mkdir $long_path
% set short_name [file attributes $long_path -shortname]
→ /tmp/tcl-book/ALONGD~1
% glob /tmp/tcl-book/ALONG* ❶
% glob $short_name ❷
→ C:/tmp/tcl-book/ALONGD~1
```

❶ Pattern `*` will not be matched against the short name version of file name.
❷ However, an exact match against the short name with no wildcard patterns will succeed.

One might consider this a quirk of the implementation.

12.3.5.5.3. Enumerating hidden files

You might expect the command `glob *` to return all files in the current directory. It does not. In particular, by default the `glob` return value does not include any “hidden” files where the term has a platform-dependent meaning.

On Windows platforms, hidden files are those that have the hidden file attribute set. On Unix, hidden files are those whose names begin with a period (`.`). On either platform, the option

[3] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

`-types hidden` (Table 12.7) must be specified. Furthermore, when this option is used, **only** hidden files will be included in the returned list.

Thus (for example) to get a list of all files in a directory, one must `concat` both the lists.

```
% concat [glob C:/tmp/tcl-book/*] [glob -types hidden C:/tmp/tcl-book/*]
→ {C:/tmp/tcl-book/a long directory name} C:/tmp/tcl-book/foo.txt C:/tmp/tcl-bo...
```

On Unix platforms, you can also use the `.*` pattern to return hidden files in which case you may also retrieve all files with the following single command.

```
glob * .*
```

There is one additional complication with hidden files on Unix. The `.*` pattern as well as the `-types hidden` option cause the special directory entries `.` and `..` to be returned. In most cases, you will want to filter these out.

12.3.6. Links: file link, file readlink

```
file link LINK
file link ?-symbolic|-hard? LINK TARGET
file readlink LINK
```

Some file systems support *hard links* where a new directory entry is created for a file, in effect giving the file another alternative name by which it can be accessed. Modification through one name will also be reflected when the file is accessed through another name. Hard links are not distinguishable from the “original” name of the file and Tcl commands do not (and cannot) distinguish between the two either.

File systems may also support the concept of *soft links* (also called *symbolic links*), where the directory entry points to a file whose content is a reference to another file, the link target. When passed an argument that is a symbolic link, some commands operate on the link itself, others operate on the target. These specifics are discussed in the description of each command. Here we only discuss commands that operate specifically on links.

Unix and Unix-like platforms support both hard and soft links to files and directories and correspondingly Tcl supports both. Although newer Windows versions support both types of links, older versions only supported soft links to directories and hard links to files and Tcl on Windows support for links is likewise limited to the same.

The first form of the `file link` command returns the target path referenced by the argument *`LINK`*. If *`LINK`* is not a path to a symbolic link, an error is generated.

The second form allows creation of a link with path *`LINK`* to the file or directory specified by *`TARGET`*. If either `-symbolic` or `-hard` is specified, the created link is of soft (symbolic) or hard type respectively. Otherwise, the command chooses a link type that is appropriate for the platform and file systems. On Unix platforms, if *`TARGET`* is a relative path, it is referenced as-is and will be interpreted by the system as relative to *`LINK`*. On other platforms, *`TARGET`* is normalized to an absolute form and *`LINK`* is set up to point to the normalized *`TARGET`* path.

The `file readlink` command returns the target of a symbolic link.

Some examples illustrating difference between directory and file links on Windows:

```
% file link /tmp/tcl-book/dirlink /tmp/tcl-book/foodir
→ /tmp/tcl-book/foodir
% file readlink /tmp/tcl-book/dirlink ❶
→ C:\tmp\tcl-book\foodir
% file link /tmp/tcl-book/dirlink ❷
→ C:\tmp\tcl-book\foodir
% file link /tmp/tcl-book/filelink /tmp/tcl-book/foo.txt
→ /tmp/tcl-book/foo.txt
% file readlink /tmp/tcl-book/filelink ❸
Ø could not read link "/tmp/tcl-book/filelink": not a directory
```

❶ Succeeds because on Windows directory links are soft links
❷ Same as above
❸ Fails because on Windows file links are hard links

The last command fails because hard links cannot be read. Nevertheless we can still verify it is indeed a link by writing to it and reading back from the target path.

```
writeFile /tmp/tcl-book/filelink "Hee haw" → (empty)
readFile /tmp/tcl-book/foo.txt             → Hee haw
```

12.3.7. Temporary files: file tempfile | tempdir

```
file tempdir ?TEMPLATE?
file tempfile ?NAMEVAR? ?TEMPLATE?
```

`TEMPLATE` is an optional path template from which the temporary path is generated. This is system specific and not discussed here. See the Tcl reference documentation.

The `file tempdir` command creates a new temporary directory and returns its path. The `file tempfile` command creates a temporary file in a system-specific directory, and returns a read-write channel (Section 13.1) to it. If *`NAMEVAR`* is provided, the command will set the variable of that name to the full path to the temporary file. If *`NAMEVAR`* is **not** provided, Tcl will attempt to delete the file when the channel is closed.

```
set fd [file tempfile temppath]     → file2415495ed10
puts $temppath                      → D:/Temp/TCL8433.TMP
puts $fd "This is a temporary file" → (empty)
close $fd                           → (empty)
readFile $temppath                  → This is a temporary file
file delete $temppath               → (empty)
```

13

Channels and Basic I/O

> Nay, be a Columbus to whole new continents and worlds within you, opening new channels, not of trade, but of thought.
>
> — Henry David Thoreau

In accordance with the above counsel, Tcl offers *channels* as a mechanism for communication with the outside world through files, sockets, and other devices as well as within an application using reflected channels. Channels provide a uniform and consistent interface to input and output of data across all device types. Furthermore, the channel abstraction provides a framework for additional functionality such as encoding and transforms.

This chapter covers basic input-output operations using files. Advanced topics are delegated to Chapter 21. Network communication and serial ports are covered in Chapter 22.

13.1. Channels and File I/O

The general sequence of steps for I/O from Tcl is

- Open a channel for reading and/or writing.
- Optionally configure the channel parameters such as encoding and buffer sizes.
- Read and write to the channel.
- Close the channel to release resources when done.

The `chan` ensemble command implements all the above with the exception of commands that open channels. Many of these also have equivalent standalone commands for historical reasons, for example, `chan read` and `read`.

13.2. Standard channels: stdin, stdout, stderr

When a process starts up, most operating systems create streams for reading and writing data. These are commonly known as *standard input*, from which data is read, *standard output* where data is written, and *standard error*, where error messages are written. Generally, the underlying device is a terminal or a pipe.

Correspondingly, Tcl creates three standard channels at startup, named `stdin`, `stdout` and `stderr` respectively. Commands that read and write data operate on `stdin` and `stdout` if a channel is not explicitly specified. The two statements below are equivalent.

```
puts foo
puts stdout foo
```

With the exception noted below for Windows, these channels can be used in the same exact fashion as channels explicitly opened by the application. If a standard channel is closed, the very next channel that is opened is assigned to that standard channel as illustrated below.

```
% chan names ❶
→ stdout stdin stderr
% close stderr
% chan names
→ stdout stdin
% set ch [open error.log w] ❷
→ stderr
% chan names
→ stderr stdout stdin
```

❶ Returns the list of currently open channels.
❷ A newly opened channel is assigned to `stderr` as it does not exist.

Closing and reopening `stdout` in a similar fashion is one way you can change where commands like `puts` write by default.

Standard channels on Windows

Windows applications may be console-mode programs that run in a command shell or have a graphical user interface (GUI). For console-mode programs, Windows creates standard channels in a manner similar to Unix systems.

For GUI programs however, Windows itself does not create standard channels as all interaction is expected to be graphical. Thus in GUI programs like `wish`, Tcl creates "pseudo" channels via the `console` command in Tk that emulate the standard channels. However this emulation is only partial since more advanced features like asynchronous I/O will not work with these emulated channels.

13.3. Creating file channels: open

```
open PATH ?ACCESS? ?PERMISSIONS?
```

The `open` command returns a channel for performing I/O operations. When no longer needed, the returned channel must be passed to the `close` command (Section 13.4) so that associated resources can be released.

The *`PATH`* argument may identify a file, process pipeline or a serial port. In this chapter, only file channels are covered deferring the others to later chapters.

The optional *`ACCESS`* argument specifies the desired access to the file. If it is not specified, or if it indicates the file is to be opened only for reading, *`PATH`* must reference an existing file.

`ACCESS` may take one of two forms — a string or a list of flags. The possible values for the string form are shown in Table 13.1.

Table 13.1. Access modes for open - string form

Mode	Description
`r`, `rb`	The file is opened only for reading in text and binary modes (Section 13.14) respectively. *`ACCESS`* defaults to `r`.
`r+`, `rb+`, `r+b`	The file must exist and is opened for both read and write access, in text mode for `r+` and binary mode for `rb+`, or equivalently `r+b`.
`w`, `wb`	The file is opened only for writing in text and binary mode respectively. It will be created if necessary. Content of existing files will be truncated.
`w+`, `wb+`, `w+b`	The file is opened for both reading and writing. It will be created if necessary and truncated if it exists. The value `w+` specifies text mode and the others binary.
`a`, `ab`	Similar to `w`, `wb` except that all writes to the file are appended to the content irrespective of the current file pointer.
`a+`, `ab+`, `a+b`	Similar to `w+`, `wb+`, `w+b` except that all writes to the file are appended to the content irrespective of the current file pointer.

The second form for the *`ACCESS`* argument is a list whose elements are flag values from Table 13.2. Exactly one of `RDONLY`, `WRONLY` or `RDWR` must be present in the list. The other flags are optional.

Table 13.2. Access modes for open - list form

Mode	Description
`RDONLY`	Open the file only for reading.
`WRONLY`	Open the file only for writing.
`RDWR`	Open the file for reading and writing.
`APPEND`	All writes are appended to the end of the file. Note this must be specified **in addition** to `WRONLY` or `RDWR`.
`BINARY`	All I/O is to be done in binary mode. Default is text mode.
`CREAT`	The file is created if necessary. Without this flag an error is raised on attempts to open non-existent files.
`NOCTTY`	The opened file is not to be made the controlling terminal for the process. Only relevant if *`PATH`* corresponds to a terminal.
`NONBLOCK`	Prevents the process from blocking while opening the file. The exact semantics are depedent on both the system and type of device. See the description of `O_NONBLOCK` in the platform documentation of the `open` system call.
`TRUNC`	Specifies that if the file exists, it is to be truncated.

The following examples illustrate various modes with which files can be opened using both string and list forms of *`ACCESS`*.

Note all channels are closed after use with the `chan close` or `close` commands.

Create a new file and write a line to it:

```
% set chan [open /tmp/tcl-book/newfile.txt w] ❶
→ file24154e03dc0
% puts $chan "Line one"
% gets $chan ❷
Ø channel "file24154e03dc0" wasn't opened for reading
% close $chan
```

❶ Open for write, truncating if it exists.
❷ Fails because not open for read.

Open an existing file for reading and writing:

```
set chan [open /tmp/tcl-book/newfile.txt r+] → file241551b4e90 ❶
read $chan                                   → Line one
close $chan                                  → (empty)
set chan [open /tmp/tcl-book/newfile.txt w+] → file24154d75a60 ❷
read $chan                                   → (empty) ❸
puts $chan "Line one again"                  → (empty)
close $chan                                  → (empty)
```

❶ Open for read and write **without** truncating.
❷ Open for read and write. Will truncate file.
❸ Notice empty string returned as file truncated.

Append to a file:

```
% set chan [open /tmp/tcl-book/newfile.txt a] ❶
→ file2415497c640
% puts $chan "Line two"                       ❷
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt] ❸
→ file24154e177e0
% read $chan
→ Line one again
  Line two
% close $chan
```

❶ Open for append. Will **not** truncate file.
❷ Line will be written at the end of the file.
❸ Open for read only (default).

On Windows systems, an attempt to open a file with the `w` or `w+` modes will fail if the file has the `hidden` or `system` attributes set. To get around this, you can either reset those attributes with the `file attributes` command before opening the file, or open the file using the `r+` mode and then truncate the file with `chan truncate`.

The following examples illustrate the use of the second form of the *ACCESS* argument that correspond to the examples for the first form above.

Create a new file and write a line to it:

```
% set chan [open /tmp/tcl-book/newfile.txt {WRONLY CREAT}] ❶
→ file24154b574e0
% puts $chan "Line one"
% gets $chan                                                ❷
Ø channel "file24154b574e0" wasn't opened for reading
% close $chan
```

❶ Open for write, creating the file if necessary.
❷ Fails because not open for read

Open an existing file for reading and writing:

```
set chan [open /tmp/tcl-book/newfile.txt RDWR]         → file24154bedf00 ❶
read $chan                                             → Line one
close $chan                                            → (empty)
set chan [open /tmp/tcl-book/newfile.txt {RDWR TRUNC}] → file24154e03dc0 ❷
read $chan                                             → (empty)
puts $chan "Line one again"                            → (empty)
close $chan                                            → (empty)
```

❶ Open for read and write **without** truncating
❷ Open for read and write. Will truncate file.

Append to a file:

```
% set chan [open /tmp/tcl-book/newfile.txt {WRONLY APPEND}] ❶
→ file24154ab8e40
% puts $chan "Line two"
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt RDONLY] ❷
→ file241551ce840
% read $chan
→ Line one again
  Line two
% close $chan
```

❶ Open for append. Will **not** truncate file.
❷ Open for read only

The *PERMISSIONS* parameter to the `open` command is only used if the file did not previously exist and has to be created. It specifies the access permissions for the newly created file together with the process' file mode creation mask. By default, *PERMISSIONS* has the value octal `0666` permitting both read and write access for all users unless limited by the process' file mode creation mask.

13.4. Closing a channel: chan close, close

```
chan close CHANNEL ?DIRECTION?
close CHANNEL ?DIRECTION?
```

When no longer needed, channels should be closed using `chan close` or, equivalently, `close`. If only a single argument is given to the command, the channel is closed for both input and output. Otherwise, *`DIRECTION`* must be `read` or `write` and the channel (presumed to be bidirectional) is only "half-closed" with no further operations of the specified type, read or write, permitted. We will see examples of half-closes when we discuss process pipelines (Section 20.4).

When a channel is closed for input, any input data not read by the application is discarded. When closed for output, all output data buffered internally by Tcl is written out to the file (or pipe, socket etc. as the case may be). If the channel is a blocking channel, the command only returns once the data has been written out and the operating system file descriptor or handle has been closed. For non-blocking channels, the command returns immediately and flushing of data and closing of handles happens in the background.

Tcl will not automatically flush any non-blocking channels that are open when the process exits without explicitly closing the channels. See the `close` reference documentation for methods to achieve this.

13.5. Channel configuration: chan configure, fconfigure

```
chan configure CHANNEL
chan configure CHANNEL OPTION
chan configure CHANNEL OPTION VALUE ?OPTION VALUE?
fconfigure CHANNEL
fconfigure CHANNEL OPTION
fconfigure CHANNEL OPTION VALUE ?OPTION VALUE?
```

Channels have configuration options that can be retrieved and set with the `chan configure` or, equivalently, `fconfigure` commands. When passed a single argument, the result is a dictionary containing the configuration for the channel.

```
% chan configure stdout
→ -blocking 1 -buffering line -buffersize 4096 -encoding utf-8 -eofchar {} -pro...
```

If two arguments are passed, the second must be a configuration option name. The result is the value of that option.

```
chan configure stdout -buffersize → 4096
fconfigure stdin -encoding        → utf-16
```

In the final form, one or more option and value pairs may be specified. The corresponding configuration options for the channel are then set to the new values.

This chapter will cover the options common to all channel types. Options that specific to a channel type will be described in the chapter dedicated to that type.

13.6. Writing to channels: chan puts, puts

```
chan puts ?-nonewline? ?CHANNEL? DATA
puts ?-nonewline? ?CHANNEL? DATA
```

The commands `chan puts`, and functionally equivalent `puts`, write data to a channel.

If *`CHANNEL`* is not specified, the channel defaults to the standard output channel `stdout` (Section 13.2). An additional newline character is appended to the output data by default. The `-nonewline` option can be passed to prevent this.

```
% puts "This will go to the standard output"
→ This will go to the standard output
% set chan [open /tmp/tcl-book/myfile.txt w]
→ file24154ace450
% puts $chan "Line one"
% puts -nonewline $chan "This is "
% puts $chan "the second line"
% close $chan
% readFile /tmp/tcl-book/myfile.txt
→ Line one
  This is the second line
```


If the `-nonewline` option is specified when writing to channels that are **line buffered**, such as standard output or standard error to a terminal, you will need to flush (Section 13.6.1) the channel for the output to show up on the device.

The data passed to `puts` or `chan puts` may not necessarily be exactly what is written to the file or output device for a number of reasons:

- The channel is configured to do end of line translation (Section 13.10).
- The channel is not in binary mode (Section 13.14).
- The channel has transforms applied (Section 21.2).

We will explore all these possibilities as we go along.

13.6.1. Output buffering

Data written to a channel is buffered and not necessarily written out to the file or device immediately. Various channel options and the `flush` command (Section 13.6.1.2) control when the data is actually written.

13.6.1.1. Buffering mode: -buffering

The `-buffering` option to `chan configure` controls when data is written out from the channel buffers to the file or other device. The various values of this option and their effect is shown in Table 13.3.

Table 13.3. Buffering policy option values

Value	Description
`none`	Buffering is disabled. Any `puts` invocation results in the data being written to the device. Note the operating system may do its own buffering.
`line`	Data is flushed from the buffer whenever a newline character is written to the channel.
`full`	Data is fully buffered and flushed when the buffer is full.

By default, all channels are configured for full buffering except for terminal-like devices which are configured to be line buffered. The `stdout` and `stderr` standard channels are initially set to `line` and `none` respectively.

```
chan configure stdout -buffering      → line
chan configure stdout -buffering none → (empty) ❶
```

❶ Reset standard output to flush on every write.

13.6.1.2. Flushing buffers: chan flush, flush

```
chan flush CHANNEL
flush CHANNEL
```

In addition to the automatic flushing of data from output buffers as described above, an application can also explicitly force a channel's buffer to be flushed with the `chan flush` or `flush` commands. All buffered output for the channel *`CHANNEL`* is written to the underlying device.

13.6.1.3. Sizing buffers: -buffersize

The `-buffersize` channel option controls the size of the buffer maintained for a channel. For bidirectional channels, this configuration setting applies to both input and output buffers.

```
chan configure stdout -buffersize   → 4096 ❶
fconfigure stdout -buffersize 10000 → (empty) ❷
```

❶ Retrieves the current buffer size
❷ Set the buffer size to 10,000 bytes

13.7. Reading from channels

Tcl provides two ways to read data from a channel, a line at a time, or a specified number of characters, which may be the entire content.

13.7.1. Reading lines from a file: chan gets, gets

```
chan gets CHANNEL ?VARNAME?
gets CHANNEL ?VARNAME?
```

The `chan gets` and the equivalent `gets` commands retrieve a line at a time from a channel. Here we describe only the blocking mode operation postponing discussion of non-blocking mode to Section 21.1.

In the single argument form, the commands return a complete line from the specified channel not including the end-of-line character sequence.

```
set chan [open /tmp/tcl-book/myfile.txt] → file241549e2ad0
gets $chan                               → Line one
```

If a second argument is specified, the read line is stored in a variable of that name and the number of characters is returned as the command result.

```
gets $chan line → 23
puts $line      → This is the second line
```

If the end of the file is reached before finding a end of line sequence, the remaining characters are returned as a complete line. If an end of line is not found and it is not the end of the channel content, for example when reading from a network socket where more data is expected, the command will block (assuming a blocking mode channel) until additional data containing an end of line arrives on the channel or the channel is closed from the remote end. Note this situation does not arise when reading from files.

After the last line is read from the channel, subsequent calls will return an empty string if the *`VARNAME`* argument is not specified. Note this cannot be distinguished from an empty line and the `chan eof` command (Section 13.7.3) must be used to distinguish between the two cases. On the other hand, if *`VARNAME`* is specified, the two cases are immediately distinguished as the end of file condition will result in the command returning `-1` versus `0` for an empty line.

```
gets $chan line → -1
```

13.7.2. Reading characters from a file: chan read, read

```
chan read ?-nonewline? CHANNEL
chan read CHANNEL NUMCHARS
read ?-nonewline? CHANNEL
read CHANNEL NUMCHARS
```

Unlike `chan gets`, `chan read` and `read` return a specified number of characters from a channel without any regard for line endings. Again, the command behaviour differs between blocking and non-blocking mode and here we only describe the former.

In the first form of the command, the commands return all data from the channel until the end of file is reached. If the `-nonewline` option is specified, the last character read is discarded if it is a newline character.

The second form reads only the number of characters passed as *`NUMCHARS`*. In this case, the command returns *`NUMCHARS`* characters from the channel unless it reaches the end of the file first, in which case it returns all the remaining characters. If there are fewer than *`NUMCHARS`* characters available in the channel and the end of the file has not been reached, the command will block (assuming blocking mode is enabled). This situation does not occur with file-based channels.

Because of various data transforms that can happen as part of I/O, the character count is not necessarily the same as the number of raw bytes read from the file or device.

Some simple examples of reading characters:

```
% set chan [open /tmp/tcl-book/myfile.txt]
→ file24154b57b20
% read $chan 1 ❶
→ L
% read $chan  ❷
→ ine one
  This is the second line
% read $chan  ❸
% eof $chan
→ 1
```

❶ Read a single character
❷ Read remaining data
❸ End of file reached (empty string returned)

13.7.3. Detecting end of file: chan eof, eof

```
chan eof CHANNEL
eof CHANNEL
```

There are several situations where an application needs to explicitly check if a channel is at end of file. We mentioned one such earlier where there is ambiguity in one form of the `chan gets` command between an empty line and end of file. Other situations arise when working with non-blocking channels where there is need to distinguish between end of file and data not being available yet.

The `chan eof` and `eof` commands check for the end of file condition on a channel. We saw an example in the previous section and we will see less trivial examples when we discuss non-blocking I/O in later chapters.

The `chan eof` and `eof` command will return `1` on an end of file condition only *after* an attempt has been made to read beyond the last character. Thus it should be checked only when `gets` or `read` indicate a potential end of file condition.

13.7.4. Input buffering

Like the output side, the input side is also buffered. However, as there is no meaningful flush operation on input, only the `-buffersize` (Section 13.6.1.3) option affects input.

13.8. File utilities: writeFile, readFile, foreachLine

Tcl provides some convenience commands encapsulating some common sequences of file operations. Because their focus is on simplicity and convenience, these commands use the default settings for a channel and do not offer the full flexibility of channel configuration for encodings, line endings etc.

13.8.1. A utility to write files: writeFile

```
writeFile PATH ?text|binary? DATA
```

The `writeFile` command stores the passed content in a file, overwriting or creating it as necessary. If the second argument is `binary`, the data is treated as binary data and written to the file verbatim. Otherwise, it is treated as text and undergoes encoding and line ending translation as per default channel settings.

```
% writeFile /tmp/tcl-book/myfile.txt "Line one\nLine two"
```

13.8.2. A utility to read files: readFile

```
readFile PATH ?text|binary?
```

The `readFile` command reads the entire content of a file. The command returns the content of the specified file as text or binary data depending on the second argument which defaults to `text`.

```
% readFile /tmp/tcl-book/myfile.txt
→ Line one
  Line two
```

13.8.3. Iterating over lines: foreachLine

```
foreachLine VARNAME PATH SCRIPT
```

The `foreachLine` command executes *`SCRIPT`* for each line in a file after assigning the line to the variable *VARNAME*.

```
foreachLine line /tmp/tcl-book/myfile.txt {
    puts [string toupper $line]
}
→ LINE ONE
  LINE TWO
```

The standard loop related control commands such as `break`, `continue` and `return` may be used within the script.

13.9. Terminal configuration

Some channel configuration is specific to terminal devices on Unix and the console device on Windows.

13.9.1. Input character processing: -inputmode

The `-inputmode` option configures the input processing mode for channels that wrap TTY's on Unix and the console on Windows, usually accessed as `stdin`. The option can take on one of the values shown in Table 13.4.

Table 13.4. Option -inputmode values

Value	Description
`normal`	Terminal or console is in normal line-oriented input mode with standard editing enabled.
`password`	Echoing of characters disabled.
`raw`	Echoing and editing disabled with all input passed directly to Tcl.
`reset`	(Write-only) Terminal or console is set to the initial state when the channel was opened.

13.9.2. Output screen size: -winsize

The read-only option `-winsize` returns a list containing the width and height in characters of the terminal or console, generally attached to `stdout` and/or `stderr`.

```
% fconfigure stdout -winsize
→ {67 54}
```

13.10. Newline translation: -translation

Internally, Tcl uses the linefeed character ASCII 10 (LF), `\n` as the newline character. This is also the convention followed on Unix platforms. On Windows however, newlines are represented by the sequence carriage return (CR) ASCII 13, `\r` followed by LF. Tcl's channel

implementation provides for the various conventions through the `-translation` option to the `chan configure` command.

```
chan configure stdout -translation → crlf
chan configure stdin -translation  → auto
```

The option value must be a one or two element list consisting of values shown in Table 13.5. The first element of the list applies to the input side of a channel. If a second element is present, it applies to the output side. If the list has only one element, it applies to both.

Table 13.5. Option -translation values

Value	Description
`auto`	On the input side, a setting of `auto` causes any occurence of the LF by itself, CR by itself, or a CRLF pair to be converted to the LF character. On the output side, the translation depends on the platform and channel type. On all channel types on Windows platforms, **and sockets on all platforms**, newlines are output as CRLF pairs. In all other cases, a single LF is output.
`cr`	On input, CR characters are treated as new lines and converted to Tcl's internal LF based newlines. On output, the reverse conversion is done.
`lf`	The external representation matches Tcl's representation and thus no conversions are performed for new lines.
`binary`	Sets the translation mode to `lf` **and** other channel options to support binary data I/O (Section 13.14).

The following sets line endings to follow the Windows-style CRLF convention.

```
chan configure $chan -translation crlf → (empty)
```

13.11. The end of file character: -eofchar

Some systems use a special end-of-file (EOF) character, for example Ctrl-Z on Windows, to mark the end of data in a file. Channels can be configured to treat the appearance of this character as end of file by setting the `-eofchar` option. The option value should be an ASCII character (other than \x00) or an empty string. Setting it to an ASCII character designates it as the EOF marker when reading from the channel. Setting it to an empty string restores the default behavior, where no character is treated as the end of file.

```
writeFile /tmp/tcl-book/eofchar.txt "abcdEfghi" → (empty)
set chan [open /tmp/tcl-book/eofchar.txt]       → file2415497b780
chan configure $chan -eofchar E                 → (empty) ❶
chan read $chan                                 → abcd ❷
chan eof $chan                                  → 1
```

❶ Set `E` as the EOF character.
❷ Reading all data from channel only returns characters before `E`.

The setting has no effect on the output side. Any EOF character needed by platform convention must be explicitly written.

Tcl 8 differs from Tcl 9 in handling of EOF characters.

- The `-eofchar` default is dependent on platform and channel type.
- EOF character affect output as well. Writing an EOF character marks the end of the file.
- The option accepts a list of up to two elements allowing different EOF character for the two directions.

13.12. Channel encoding: -encoding

We saw in Section 9.1 that Tcl strings need to be encoded as a sequence of physical bytes when storing to files or communicating with other programs.

To revisit the example there, consider the Portugese word Olá. If we wanted to write this to a file that was to be read by another program that expected the content to be in CP860 (Portugese code page) encoding, we would have to first convert the string to a byte sequence in that encoding before writing to the file.

Having to explicitly encode every string before writing to a file is inconvenient and prone to errors. Instead we can configure the channel with the `-encoding` option to automatically do the conversion on every I/O operation. Any encoding name returned by `encoding names` can be passed as the option value.

```
set greeting "\u004f\u006c\u00e1"             → Olá
set chan [open /tmp/tcl-book/portugese.txt w] → file24154cdd760
chan configure $chan -encoding cp860          → (empty)
puts $chan $greeting                          → (empty)
close $chan                                   → (empty)
```

Although the above example shows output to a channel, the encoding applies to input as well. Data read from the channel will be expected to be in CP860 encoding and decoded into a Tcl string.

In Tcl 8, the `-encoding` option accepted `binary` as a synonym for the `iso8859-1` encoding. This is no longer supported in Tcl 9.

13.13. Encoding profiles: -profile

Conversion of encoded byte streams to and from Tcl strings may result in errors during encoding, when a character is not representable in the target encoding, as well as during decoding, when invalid bytes are present in the byte stream. Encoding profiles (Section 9.1.1) control how these errors are handled when doing explicit conversion with the `encoding` command. These profiles may likewise be configured for a channel by setting the `-profile` option. This determines how encoding errors are handled on the channel.

With the exception of `stderr`, channels default to the `strict` profile so any encoding errors on input will raise an error. The `stderr` channel defaults to the `replace` profile to ensure error messages can always be output even when incorrectly encoded.

As an illustrative example, the following attempt to read our CP860 file without configuring the encoding for the channel will result in an error.

```
% set chan [open /tmp/tcl-book/portugese.txt]
→ file24154a775d0
% read $chan
Ø error reading "file24154a775d0": invalid or incomplete multibyte or wide char...
% close $chan
```

Setting the profile to `replace` permits reading of the channel with invalid encoded byte sequences to be replaced by the Unicode `�` (U+FFFD) replacement character.

```
% set chan [open /tmp/tcl-book/portugese.txt]
→ file24154b6a600
% chan configure $chan -profile replace
% read $chan
→ Ol�
% close $chan
```

Of course, setting the correct encoding would be ideal but that is not always known.

```
% set chan [open /tmp/tcl-book/portugese.txt]
→ file24154d50c40
% chan configure $chan -encoding cp860
% read $chan
→ Olá
% close $chan
```

Encoding profiles are not available in Tcl 8.6 and earlier. Channel I/O behaves as though the `tcl8` profile was configured for the channel.

13.14. Binary I/O

Much of Tcl's channel system assumes files contain text content and the automatic translation of line endings, encodings etc. are directed towards convenient and portable text I/O. When reading or writing binary data (Chapter 8) however, we need to turn off these features.

- The channel encoding should be `iso8859-1` which essentially maps each byte to the Unicode code point with the same numerical value.
- Line endings should be the LF character.
- The EOF character should be disabled.

In the case of files, this can be accomplished by including the `b` qualifier in the access mode string (Table 13.1) or `BINARY` in its list form (Table 13.2). Compare the configuration values for `-translation` and `-encoding` below for channels opened with and without the `b` qualifier.

```
set txt_chan [open /tmp/tcl-book/myfile.txt r]  → file24154cdd760
set bin_chan [open /tmp/tcl-book/myfile.txt rb] → file24154f21c30
chan configure $txt_chan -encoding              → utf-8
chan configure $bin_chan -encoding              → iso8859-1
chan configure $txt_chan -translation           → auto
chan configure $bin_chan -translation           → lf
```

Channels can also be set to binary mode by passing `binary` as the value for the `-translation` option. This is useful when the channel creation command, for example `socket`, does not have a means to specify binary mode or on channels where applications may need to switch between text and binary modes. Thus the above example may also be written as

```
set chan [open /tmp/tcl-book/myfile.txt r] → file24154d6a630
chan configure $chan -encoding             → utf-8
chan configure $chan -translation          → auto
chan configure $chan -translation binary   → (empty)
chan configure $chan -encoding             → iso8859-1
chan configure $chan -translation          → lf
```

Note how setting the `-translation` option to `binary` actually sets its value to `lf` and also changes the `-encoding` option.

The `chan isbinary` command checks if a channel is set up for binary I/O.

```
chan isbinary $chan  → 1
chan isbinary stdout → 0
```

The `chan isbinary` command is not available in Tcl 8.6 and earlier. The setting for the affected options above need to be explicitly checked.

13.15. The file access pointer

Every channel has an associated file access pointer that tracks the current position in the file. Any subsequent reads and writes occur starting at this position. The pointer is then updated to the offset just after the read or write. This pointer can be read and set with the `chan tell` and `chan seek` commands.

13.15.1. Retrieving the file access pointer: chan tell, tell

```
chan tell CHANNEL
tell CHANNEL
```

The `chan tell` and `tell` commands return the value of the file access pointer. For channels that do not support this operation, the command returns `-1`.

The following example shows how the file pointer is moved with each I/O operation.

```
set chan [open /tmp/tcl-book/myfile.txt] → file241549d44d0
chan tell $chan                          → 0
gets $chan                               → Line one
chan tell $chan                          → 10
gets $chan                               → Line two
chan tell $chan                          → 18
```

It is important to note that the return value from these commands is an offset in **bytes, not in characters**, from the beginning of the file. The difference arises from multi-byte encoding and end of line translations.

13.15.2. Setting the file access position: chan seek, seek

```
chan seek CHANNEL OFFSET ?ORIGIN?
seek CHANNEL OFFSET ?ORIGIN?
```

The `chan seek` and `seek` commands change the file pointer so that the next I/O operation begins at a different position from where the last one ended. The *`OFFSET`* argument must be an integer, positive or negative. The file pointer will be moved by these many bytes from the position specified by *`ORIGIN`* which should be one of the values from Table 13.6.

Table 13.6. Origin values for seek

Origin	Description
`start`	*OFFSET* is with respect to the start of the file and so effectively an absolute offset into the file. This is the default.
`current`	*OFFSET* is with respect to the current file pointer position.
`end`	*OFFSET* is with respect to the end of the file and usually negative.

For channels that do not support the seek operation, an error is raised.

The following example illustrates the use of `seek` and `tell`.

```
set chan [open /tmp/tcl-book/myfile.txt] → file24154eacc10
gets $chan                               → Line one
set pos [chan tell $chan]                → 10 ❶
gets $chan                               → Line two
chan seek $chan $pos                     → (empty) ❷
gets $chan                               → Line two
close $chan                              → (empty)
```

❶ Note position of second line.
❷ Return to beginning of second line.

The next example overwrites the last few characters from a file.

```
% set path /tmp/tcl-book/seek.txt
→ /tmp/tcl-book/seek.txt
% writeFile $path "1234567890"
% set chan [open $path r+b]
→ file24154adc5d0
% chan seek $chan -5 end ❶
% puts -nonewline $chan abc
% chan seek $chan 0 end
% puts -nonewline $chan def
% close $chan
% readFile $path
→ 12345abc90def
```

❶ Note the negative offset

> ! When a channel is configured for binary I/O, you can use any integer values for the *OFFSET* argument since there is a one-to-one correspondence between the file position pointer and read/write counts. In text mode however, this correspondence does not hold and the offsets specified should be either a value returned by `tell` or `0`.

13.16. Truncating files: chan truncate

```
chan truncate CHANNEL ?LENGTH?
```

The `chan truncate` command truncates the file or other data stream open in a channel to a specific number of bytes (**not characters**). If the *LENGTH* argument is specified, the file or data stream length is set to that value. Otherwise, the file is truncated at the current file pointer value for the channel.

13.17. Copying data between channels: chan copy, fcopy

```
chan copy FROMCHAN TOCHAN ?-size SIZE? ?-command CALLBACK?
fcopy FROMCHAN TOCHAN ?-size SIZE? ?-command CALLBACK?
```

While data may be copied between channels with `read` and `puts`, a more efficient method is to use the `chan copy` or `fcopy` commands.

The advantage of `chan copy` over the `read` and `puts` method is primarily efficiency. It minimizes both CPU and memory usage by avoiding buffer copies.

If the `-size` option is present, only the number of bytes specified by the option are copied from the input channel *FROMCHAN* to *TOCHAN*. Otherwise, all data until end of file is copied.

Without the `-command` option, the commands will block until the copy is complete. If the option is specified, the commands will return immediately and the data copying will continue

in the background via the event loop. On completion of the copy, the callback command is invoked. Note that the event loop (Chapter 19) must be running if this option is passed.

The command respects the encoding and translation settings of each channel. The following will convert the CP860 encoded file to one using UTF-8.

```
set from [open /tmp/tcl-book/portugese.txt]
chan configure $from -encoding cp860
set to [open /tmp/tcl-book/portugese.utf8 w]
chan configure $to -encoding utf-8
chan copy $from $to
close $from
close $to
```

13.18. Enumerating open channels: chan names

The `chan names` command returns the list of currently open channels.

```
chan names ?PATTERN?
```

The command returns the list of channels with names matching *`PATTERN`* which has the same syntax used in the `string match` (Section 4.24) command. All channels names are returned if the *`PATTERN`* argument is not supplied.

```
chan names       → stderr stdout stdin rc13
chan names stdo* → stdout
```

14

Code Execution

Tcl is distinguished by the flexibility and versatility of its execution model that makes it amenable to be used in a wide variety of software architectures and patterns. In this chapter, we will study this model as background for the more sophisticated material in later chapters dealing with events and coroutines. We will also cover treatment of code as data, allowing it to be constructed and manipulated at runtime, a concept known as *metaprogramming*.

14.1. Frames and the call stack

As is true for practically all programming languages, Tcl has to keep track, at each stage of a computation, of the execution context to be used for resolving names of commands and variables. An understanding of the structures underlying an execution context is a prequisite to effectively manipulating them within a script.

14.1.1. The call stack

Consider the following program

```
namespace eval areas {
    variable pi 3.142
    proc circle {radius} {
        variable pi
        set area [expr {$pi*$radius*$radius}]
        return $area
    }
}
areas::circle 2
→ 12.568
```

When Tcl begins execution, it does so in the *global* context where all variables and commands resolve to the global namespace (Chapter 16) unless they are explicitly qualified with a namespace. Being outside a procedure context, there are no local variables. This execution context is stored in a *call frame* as shown in Figure 14.1.

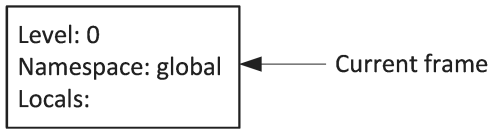

Figure 14.1. Initial call frame

When the `areas::circle` command in invoked, the `areas` relative namespace name is resolved (Section 16.5.3) in the current (global) context resulting in the `::areas::circle` procedure being called. A procedure call has (potentially) local variables as well as (again, potentially) a different namespace context. Thus a new call frame reflecting these is added on every procedure call. This collection of frames is known as the *call stack* and the *level* of each frame is its index in the stack.

While `areas::circle` is executing, the call stack then looks as shown in Figure 14.2.

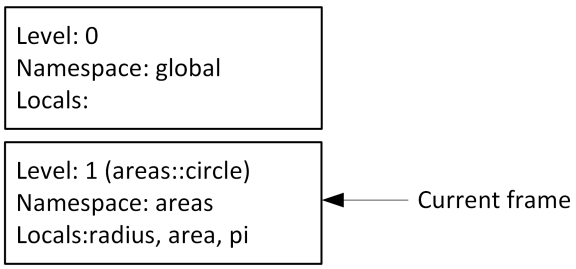

Figure 14.2. Level 1 call frame

The local variables include the argument `radius` and a procedure-local variable `area`. Any references to these variable names will be resolved from this list of locals. Similarly, the namespace context is now `areas` and correspondingly the `variable` command creates a local variable called `pi` that is linked (Section 16.3) to the variable of the same name in the `areas` namespace.

The invocation of the `expr` command on the other hand does **not** result in a new call frame. New call frames are only created by commands that may change the namespace context or have local variables, such as procedures and `namespace eval` (Section 16.1). Since `expr` does neither, like most commands, it executes in the context of its caller and does not necessitate a new call frame.

When a procedure completes execution, its call frame is *popped* off the call stack. The call stack then again looks like Figure 14.1.

The above is a simplified, not quite accurate, depiction but sufficient for our purposes.

14.1.2. Inspecting the call stack: info level

```
info level ?LEVEL?
```

The `info level` command returns the state of the call stack. If *`LEVEL`* is not present, the command returns the depth of the stack.

```
proc cmdA {} {cmdB}                                  → (empty)
proc cmdB {} {puts "cmdB info level: [info level]"} → (empty)
puts "Global info level: [info level]"               → Global info level: 0
cmdA                                                 → cmdB info level: 2
```

If the optional *LEVEL* argument is specified, the command returns a list containing the command name and arguments that were specified for the command that resulted in the creation of the call frame at that level in the call stack.

LEVEL may be

- a positive number referencing a call frame at an absolute stack level,
- `0` referencing the currently active call frame,
- a negative number, referencing the call frame relative to the current one, `-1` being the one above the current frame, and so on.

```
proc cmdA {a {b 0}} {
    puts "Level [info level]: cmdA: I was called as '[info level 0]'"
    cmdB $a
}
proc cmdB {a} {
    puts "cmdB: I was called as '[info level 0]'"          ❶
    puts "cmdB: My caller was called as '[info level -1]'" ❷
    puts "cmdB: The command invoked at the global level was '[info level 1]'"  ❸
}
cmdA 1 2
→ Level 1: cmdA: I was called as 'cmdA 1 2'
  cmdB: I was called as 'cmdB 1'
  cmdB: My caller was called as 'cmdA 1 2'
  cmdB: The command invoked at the global level was 'cmdA 1 2'
```

❶ Relative level `0`.
❷ Relative level `-1` is caller's context.
❸ Command executing at level `1`, i.e. invoked at global level.

Commands: Invoked versus executed

Note the information returned is the command that the caller invoked, not the command that is executed. What is the difference? If the procedure has optional default arguments that are not specified by the caller, they will not be included in the result of the `info level` command. Compare output below with that above.

```
cmdA 1
→ Level 1: cmdA: I was called as 'cmdA 1'
  cmdB: I was called as 'cmdB 1'
  cmdB: My caller was called as 'cmdA 1'
  cmdB: The command invoked at the global level was 'cmdA 1'
```

The information returned by `info level` makes it easy to print out the entire call stack for any procedure invocation for debugging or troubleshooting purposes. In a later section we will see how we can do this at runtime without modifying any application code.

14.1.3. Commands that create call frames

Not all commands create call frames. As stated earlier, commands that execute scripts and support locally scoped variables whose lifetime is tied to the command execution, or commands that change namespace contexts, need new call frames. These include

- Procedure calls
- Object method calls (Chapter 18)
- The `namespace eval` command (Section 16.1)
- Coroutines (Chapter 24)

On the other hand, the commands `eval` (Section 3.13), `try` (Section 15.4.3), `source` (Section 3.14) and control statements execute scripts but do not need new call frames as they do not host local variables or change namespace contexts.

It is easy enough to check whether a command adds a call frame.

```
info level                      → 0 ❶
eval {info level}               → 0 ❷
namespace eval ns {info level} → 1 ❸
apply {{} {info level}}         → 1 ❹
```

❶ Current frame level
❷ `eval` runs in the namespace of the caller and has no local variables.
❸ `namespace eval` has no local variables but changes the namespace context.
❹ Anonymous procedure calls are just procedure calls.

14.1.4. Referencing variables in call frames: upvar

```
upvar ?LEVEL? ?VARNAME LOCAL …?
```

The `upvar` command offers the ability for a procedure to reference a variable defined anywhere in its call stack, even local variables in other procedures.

The *`LEVEL`* argument specifies the level in the call stack and defaults to `1`.

- If a non-negative integer, *`LEVEL`* specifies the number of levels up the call stack that the variable to be referenced resides.
- A level of `0` references the current frame.
- If *`LEVEL`* begins with `#` immediately followed by an non-negative integer, it gives the **absolute** level in the call stack with `#0` referring to the global context.

The syntax used for the *`LEVEL`* argument differs from the syntax used for the argument to the `info level` command.

For each *`VARNAME LOCAL`* pair, the command will create a local variable *`LOCAL`* and link it to a variable *`VARNAME`* in the referenced call frame. All accesses to *`LOCAL`* will then access *`VARNAME`*.

Time for a few examples to clarify the variations. In the script below, we define a variable named `myvar` in multiple contexts and then examine the call stack to see how each is referenced.

```
set myvar "Global" ❶
proc gproc {} {
    set myvar "gproc" ❷
    upvar #0 myvar var#0 ❸
    upvar #1 myvar var#1 ❹
    upvar 1  myvar var1 nsvar nsvar ❺
    upvar 0  myvar var0 ❻
    puts "var#0 = ${var#0}, var#1 = ${var#1}, var1 = $var1, var0 = $var0"
    set nsvar "Created via linked variable"
    unset var#0
}
namespace eval ns {
    variable myvar "ns"
    proc nproc {} {
        variable nsvar
        set myvar "nproc"
        gproc
    }
}
```

❶ Global variable
❷ Local variable
❸ Local variable linked to global variable
❹ Local variable linked to variable in frame #1
❺ Local variables linked to variable in caller's context
❻ Local variable linked to variable in the current context

Now if we were to run the command

```
namespace eval ns nproc
```

the call stack when the `puts` command in `gproc` is invoked will look as shown in Figure 14.3. A few points to be noted from the figure:

- Call levels can be referenced using absolute levels (`#0`, `#1`) or relative levels (`0`, `1`).
- The referenced variables are all named `myvar` (of course, this need not be the case) but are distinguished by the fact that they all appear in different call frames or namespace contexts.
- The referenced name may be that of a global variable, a namespace variable, a local variable in a procedure on the stack or itself be a linked variable (e.g. `nsvar`)
- **The referenced variable need not actually exist** (e.g. `nsvar`). It will be created if written to. Conversely, unsetting a linked variable (`var#0`) unsets the variable to which it is linked (global `myvar`).

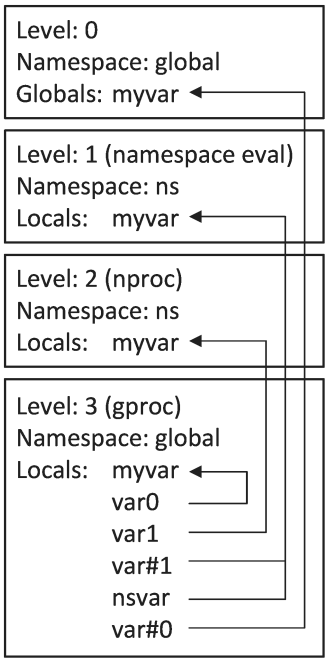

Figure 14.3. Call stack and upvar

For confirmation, we will run the command to verify the output.

```
% namespace eval ns nproc
→ var#0 = Global, var#1 = ns, var1 = nproc, var0 = gproc
% info exists ::myvar ❶
→ 0
% puts $::ns::nsvar    ❷
→ Created via linked variable
```

❶ Was unset via the linked variable.
❷ Was created via the linked variable.

Using `upvar`

Now that we know how `upvar` works, when is it actually of use?

As a first example, consider implementing a command `lprepend`, which works like the `lappend` command (Section 5.9) except that it adds an element to the front of a list contained in a variable instead of the end. The command will have the signature

```
lprepend VAR ?ELEM …?
```

where *`VAR`* is the name of a variable, which may be a local, global or a namespace variable. The remaining arguments are values to be prepended to its existing content in order.

Such a command could be implemented using `upvar`:

```
proc lprepend {varname args} {
    upvar 1 $varname var
    set var [linsert $var 0 {*}$args]
}
set lvar {1 2}
lprepend lvar 3 4
puts $lvar
→ 3 4 1 2
```

Notice here that unlike our earlier examples, **the referenced variable name is not hardcoded into the `upvar` invocation but rather is itself passed through a variable**.

Another example where this is useful is when a procedure has to modify the contents of an array. Remember that arrays are themselves variables, not values. To modify an array then, we can just pass its name to the procedure.

Here is a procedure to change values of all array elements to uppercase.

```
proc upcase_array {arrayvar} {
    upvar 1 $arrayvar arr
    foreach {key val} [array get arr] {
        set arr($key) [string toupper $val]
    }
}
array set myarr {1 one 2 two}
upcase_array myarr
parray myarr
→ myarr(1) = ONE
  myarr(2) = TWO
```

A use of `upvar` that might not be immediately obvious is as a convenience to reduce typing effort or increase readability. For example,

```
proc myproc {} {
    upvar 0 ::ns::nsvar nsvar
    upvar 1 ::myarr(0) elem
    puts $nsvar
    set elem zero
}
```

Notice that by using a *`LEVEL`* argument of `0`, we are not really changing the call frame or the variable context. We are simply creating a new name and linking it to a variable that was already available in the current context (using fully qualified names).

Variable aliases created with `upvar` cannot be used with commands like `trace` (Section 14.2) or `vwait` (Section 19.2.3.1) and with the `-textvariable` option associated with Tk widgets. You need to provide these commands with the name of the original variable instead.

14.1.5. Executing scripts in a call frame: uplevel

```
uplevel ?LEVEL? ARG ?ARG …?
```

Having looked at `upvar` which allows access to variables in any frame on the call stack, we now turn our attention to the more general purpose and powerful `uplevel` command which allows **execution** of code within the context of any frame on the stack. It is this command that underlies some of Tcl's most dynamic features such as the ability to define new control constructs that are on par with the built-in ones like `while` or `switch`.

The command is very similar to `eval` (Section 3.13) in its behaviour in that it concatenates its *`ARG`* arguments and executes the result as a Tcl script. It differs from `eval` in that it accepts the *`LEVEL`* argument which specifies the frame on the stack within whose context the constructed script is to be executed.

This similarity to `eval` (Section 3.13) also implies that care needs to be taken to properly protect against double substitution when and where appropriate. See the discussion in Section 3.13.1 for details.

The format of the *`LEVEL`* argument is the same as that for `upvar` (Section 14.1.4). If unspecified, *`LEVEL`* defaults to `1`. However, as for `upvar` it is strongly recommended that it be specified in case the first *`ARG`* matches one of the forms used to specify a level. Unlikely, but remember command names in Tcl can pretty much take on any form.

Time for some examples again. This time instead of showing the context using boring old variables within each procedure as we did for `upvar` (Section 14.1.4), we will print out the command being executed at each level. Refer back to `info level` (Section 14.1.2) if you don't understand this code.

```
proc cmdA {} { cmdB }
proc cmdB {} { cmdC }
proc cmdC {} {
    uplevel 0 {puts [info level]:[info level [info level]]} ❶
    uplevel 1 {puts [info level]:[info level [info level]]} ❷
    uplevel 2 {puts [info level]:[info level [info level]]} ❸

    uplevel #1 {puts [info level]:[info level [info level]]} ❹
}
cmdA
→ 3:cmdC
  2:cmdB
  1:cmdA
  1:cmdA
```

❶ Execute in the current frame (`cmdC` itself)
❷ Execute in caller's context
❸ Execute in frame 2 levels above
❹ Execute in context of Level 1

The call stacks when `cmdC` is running look as shown in Figure 14.4.

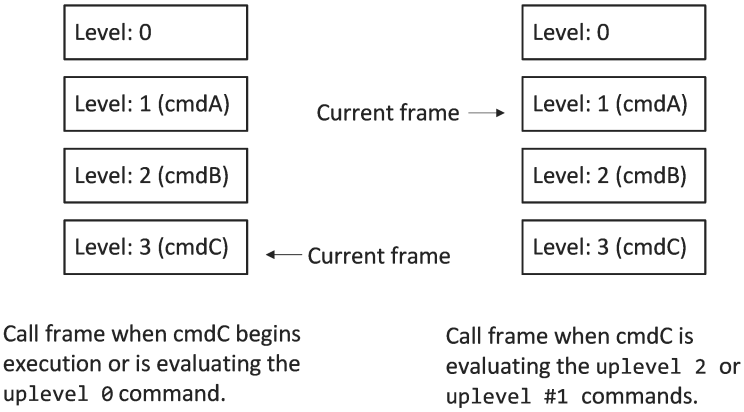

Figure 14.4. Call stack and uplevel

For the duration that `cmdC` is running, there are four frames on the call stack as shown. The *current frame* though, which holds the context to resolve unqualified variable and command names, changes through the execution of the procedure. On entry to the procedure, as well as during execution of `uplevel 0`, the current frame is the level 3 frame as we have seen before. All variables and commands will be resolved in the context of `cmdC`. On the other hand, when the uplevel commands with a level other than 0 are executed, this current frame pointer moves that many levels up the stack. Thus as shown, with `uplevel 2` the current frame will be that for `cmdA` and all names will be resolved in that context.

The most common values for *`LEVEL`* passed to `uplevel` are `#0` to execute in the global context and `1` to execute in the caller's context. Let us see a "real world" examples of each.

Use of `uplevel` to implement an interactive shell

Consider writing a "read, eval, print, loop" that allows the user to enter commands in Tcl.

```
proc repl {} {
    set command ""
    set prompt "% "
    puts -nonewline stdout $prompt
    flush stdout
    while {[gets stdin line] >= 0} {
        append command "\n$line"
        if {[info complete $command]} {
            catch {uplevel #0 $command} result
            puts stdout $result
            set command ""
            set prompt "% "
        } else {
            set prompt "(cont)% "
        }
        puts -nonewline stdout $prompt
        flush stdout
    }
}
```

Ignoring the bulk of the code which primarily concerns I/O, the key thing to note is that users expect commands to execute in the global context. This is what the `uplevel #0` command does. Had we used `eval` (Section 3.13) instead, the command would have been executed in the context of the `repl` procedure which is not what the user would expect in an interactive shell.

We have not seen the `catch` (Section 15.4.1) command as yet. For now, it suffices to know that it is used here to handle any exceptions that may be raised during the execution of the entered command.

We saw in Section 3.14.1 the means for checking if a file is being sourced as the main application. We can use that in conjunction with our `repl` procedure as follows.

```
if {[info exists ::argv0] &&
    [file dirname [file normalize [info script]/...]] eq [file dirname
 [file \
        normalize $argv0/...]]} {
    repl
}
```

You will often find similar code at the bottom of the library scripts. If the script is being sourced as an embedded module from a main application, the code is effectively disabled. On the other hand, if the file is being sourced directly from the command line as the main application, it enters the prompt loop. This is an easy means to try out commands in the library script interactively for purposes of debugging or experimentation.

Using `uplevel` to implement new control structures

Another common use of `uplevel` is to implement new control statements that exhibit all the characteristics of the ones like `while` and `switch` that Tcl provides out of the box. For instance, let us define a command `repeat` that will execute a script a given number of times. A sample use might look like

```
set sum 0
repeat i 10 {
    incr sum $i
}
```

The `repeat` command might be implemented as below

```
proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        uplevel 1 $body
    }
    return
}
```

The loop variable passed has to be updated in the caller's context so we use `upvar` (Section 14.1.4) to link to it. In addition, the loop body also has to execute in the **caller's context** so that both variable names and commands will resolve as expected. This is accomplished by the `uplevel` command as shown. We can try out our new control structure.

```
% set sum 0
→ 0
% repeat i 5 { incr sum $i }
% puts "The sum of the first $i natural numbers is $sum"
→ The sum of the first 5 natural numbers is 10
```

However, this implementation is not complete. It will not behave in the same manner as the built-in control statements in the presence of errors, `break` or `return` statements and the like within the loop body. A complete implementation will have to wait until after we discuss return codes and exception handling in Tcl.

Finding the caller's namespace

You may wish to skip this section and return to it after learning about namespaces in Chapter 16.

There is one other common use of `uplevel` that you will see in commands that themselves create new commands, for example in object frameworks. When these new commands are created, care has to be taken that their names are placed within the proper namespace context.

Let us assume we want to implement such a framework where the command `newobj` will construct a new command of a given name (we don't really care what it does) that can be used as follows.

```
newobj cmd1          ❶
newobj ns::cmd2      ❷
namespace eval ns {newobj cmd3}    ❸
namespace eval ns {newobj ::cmd4} ❹
```

❶ Should create ::cmd1
❷ Should create ::ns::cmd2
❸ Should create ::ns::cmd3
❹ Should create ::cmd4

In our simplistic example where the created commands simply print their name, all our `newobj` procedure really has to do is to ensure the command name is created in the correct namespace context:

- If the name is already fully qualified, it can be used as is.
- Otherwise, the name is relative to the **caller's** namespace so we use `uplevel` to retrieve that and qualify the constructed command with the namespace name.
- Finally, as a matter of policy we check that the name does not already exist. (It is really a matter of choice whether to allow commands to be overwritten.)

We can write this procedure as follows.

```
proc newobj {name} {
    if {[string match ::* $name]} {
        set cmdname $name
    } else {
        set ns [uplevel 1 {namespace current}]
        if {$ns eq "::"} {
            set cmdname ::$name
        } else {
            set cmdname ${ns}::$name
        }
    }
    if {[namespace which -command $cmdname] ne ""} {
        error "command $name already exists"
    }
    proc $cmdname {} "puts {I am $cmdname}"
    return
}
```

To verify our command name generation,

```
% newobj cmd1
% cmd1
→ I am ::cmd1
% namespace eval ns {newobj cmd3}
% cmd3 ❶
Ø invalid command name "cmd3"
% ns::cmd3
→ I am ::ns::cmd3
% newobj ns::cmd3 ❷
Ø command ns::cmd3 already exists
```

❶ Error. `cmd3` was not created in the global namespace
❷ Error. Command by that name already exists!

14.1.6. The internal C stack

We have talked about the call stack that keeps track of the execution contexts through a chain of procedure calls. These contexts control how variables, commands and namespaces are resolved. Tcl also maintains another stack, maintained internally and not directly visible at the scripting level, that keeps track of (among other things) the currently executing command and the location to continue from when it completes. For the lack of a better term, we will call this the internal C stack[1].

This internal C stack will become relevant when we discuss more sophisticated programming models in Tcl including recursion, the event loop and coroutines.

[1] In reality, Tcl maintains multiple internal stacks but we will not concern ourselves with that as it is an implementation detail.

To illustrate the relationship between the call stack and the internal C stack, consider execution of the following script which prints the call stack level at which each procedure is executing.

```
proc demo1 {} {
    puts "[info level [info level]]: Level [info level]"
    demo2
}
proc demo2 {} {
    puts "[info level [info level]]: Level [info level]"
    uplevel 1 {
        puts "uplevel: Level [info level]"
        demo3
    }
}
proc demo3 {} {
    puts "[info level [info level]]: Level [info level]"
}
demo1
→ demo1: Level 1
  demo2: Level 2
  uplevel: Level 1
  demo3: Level 2
```

The states of the call stack and the C stack at two stages of evaluation are shown in Figure 14.5. The left side shows the state during execution of `puts` in the `demo2` procedure while the right side shows the state during execution of `puts` in the `demo3` procedure.

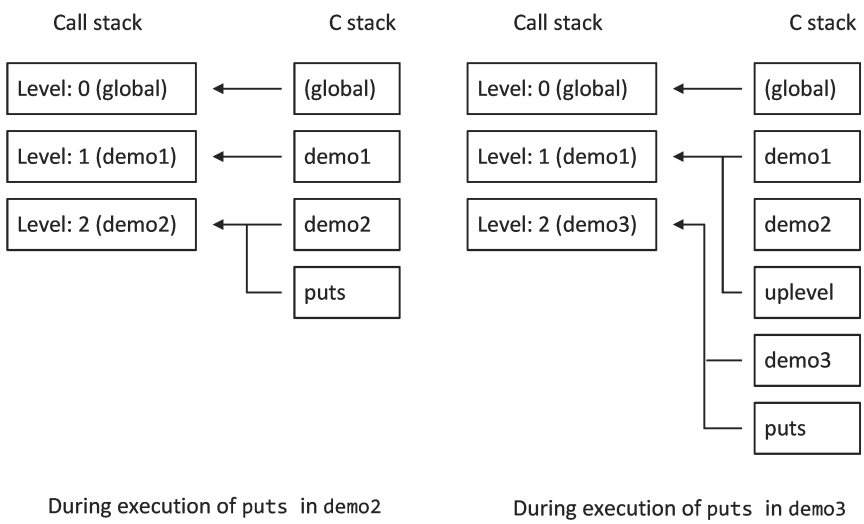

Figure 14.5. C stack and call frames

Note the following points illustrated by the figure:

- The `puts` command does not create a new call frame in the call stack as it resolves names within the context of its caller. Nevertheless it adds a slot to the C stack where Tcl stores its caller and return information.
- Likewise, the `uplevel 1` command adds a slot to the C stack as well. On the other hand, its associated context level is actually less than that of its caller.
- When `demo3` is called via `uplevel`, the context for `demo2` does not even appear on the call stack. (This does not mean it has disappeared. It simply is not accessible through the call stack until control returns to evaluation of `demo2`.)
- Notice how the depth of the C stack has grown even while the call stack depth stays the same.

This last point is the most important one to note because it impacts the maximum depth of recursive algorithms and serves as the motivation for the `tailcall` command we discuss next.

14.1.7. Recursing in place: tailcall

```
tailcall COMMAND ?ARG …?
```

The growth of the internal C stack that we described in the previous section is generally not an issue because procedure calls rarely nest deep enough for it to be a problem. The one common situation where it can be a factor is in the implementation of recursive algorithms. Let us illustrate with a simple command that calculates the sum of the first N natural numbers. We will use a recursive command instead of a simple loop because the latter would not impress anyone.

```
proc sum {n {total 0}} {
    if {$n == 0} { return $total }
    sum [expr {$n-1}] [incr total $n]
}
```

This works well enough for small numbers.

```
sum 4 → 10
```

However, see what happens when we try to sum the first 1000 integers.

```
sum 1000 Ø too many nested evaluations (infinite loop?)
```

The error you see comes from Tcl aborting the evaluation to guard against an overflow of the C stack which would lead to the process crashing. Although the `interp limit` command (Section 23.11.1) can be used to change the limit of recursion, this merely postpones the problem. Morever, changing the recursion limit with `interp limit` is dangerous without recompiling or relinking Tcl to increase the process stack size.

The problem of stack growth can be solved for certain kinds of recursive algorithms where the recursion is the last operation in the execution of the function or procedure. Under these circumstances, the context of the **calling** procedure need not be maintained because there is nothing to be done after the called procedure returns. Thus the stack space occupied by the calling procedure can be **reused** for the called procedure.

This is what the `tailcall` command effects. It invokes *`COMMAND`* passing it any supplied arguments, but instead of allocating a new context, it **overwrites the context of its caller** with that of *`COMMAND`*.

Before we go into a detailed explanation, let us rewrite our example using `tailcall`.

```
proc sum {n {total 0}} {
    if {$n == 0} { return $total }
    tailcall sum [expr {$n-1}] [incr total $n]
}
```

You can now sum without running into recursion limits.

```
sum 100000 → 5000050000
```

A look at the internal C stacks, shown in Figure 14.6, in the computation of `sum 2` in the two cases will tell us why.

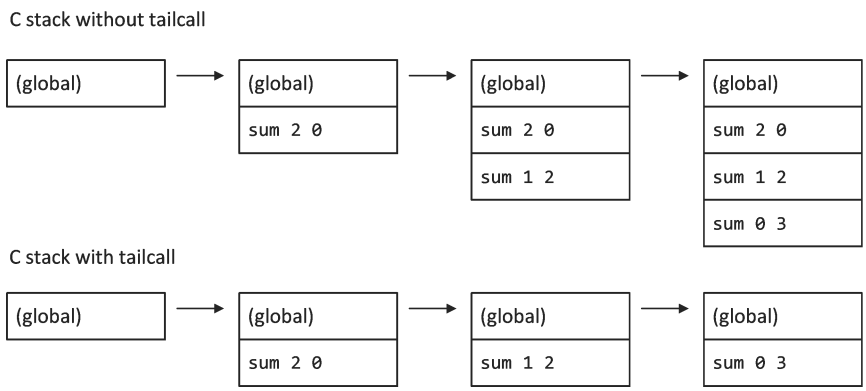

Figure 14.6. Call stack with tailcall

As `sum` recurses, the first version creates an additional C stack frame for every recursive call. The tailcall version on the other hand, reuses frames thereby keeping the number of frames on the stack constant.

In some instances, `tailcall` is useful even without any recursion being present. For instance, we saw in Section 3.5.3 a simple method for wrapping a procedure by renaming it and then calling it from the redefined procedure. That method had the drawback that it increased the call stack depth and would not work with commands like `foreach` that use `uplevel` or the equivalent to execute in their caller's context. The failure mode is demonstrated by the following example.

```
rename while _builtin_while
proc while args {
    puts "while called"
    _builtin_while {*}$args
}
set n 2
while {$n > 0} {puts $n ; incr n -1}
Ø can't read "n": no such variable
  while called
```

The command failed because our `while` wrapper executed the built-in command within its own context where there was no variable `n`. In this example, that could be fixed via an explicit `uplevel`. However, an easier and more general solution is to use `tailcall` to delegate the invocation. This way the stack depth remains the same when the original command is called and the body of the `while` is evaluated in the context of the original caller.

```
proc while args {
    puts "while called"
    tailcall _builtin_while {*}$args
}
set n 2
while {$n > 0} {puts $n ; incr n -1}
→ while called
  2
  1
```

This then works as expected. The command is similarly useful in scenarios like delegation of object methods.

Be aware that `tailcall` executes a **command**, not a **script**, so there are no issues around double substitution as there are with `eval` (Section 3.13) or `uplevel` (Section 14.1.5). If you need to call a script, as opposed to a single command, in tail-recursive fashion, use `eval` in combination with `tailcall` as below.

```
tailcall eval { Your script }
```

Or, if exception handling is required, `try` (Section 15.4.3) can be used in place of `eval` with appropriate error handling clauses specified.

```
tailcall try { Your script } on error {} { Your error handler }
```

With that introduction to `tailcall` usage under our belt, we can move on to detailing exactly what `tailcall` does.

The details behind the operation of `tailcall` are only important when you are using it in conjunction with other commands like `uplevel` (Section 14.1.5) that affect call stacks or with control structures like `try` (Section 15.4.3). For its primary use for simple recursion as in the above example, these details are not important and may be skipped.

The `tailcall` command works by

- first arranging for its command argument to be invoked after the completion of the call frame within which the `tailcall` was invoked, and then
- forcing its caller to complete immediately with a return code value of `2 / return`. **Note the return code from a command invocation is not the same as the command result.** Return codes are discussed in Section 15.2.1.

As we will see, this two step process can be a bit tricky when the call frame is not directly that of the caller but let us first start with a simple example.

```
proc demo1 {} {
    puts "demo1 enter"
    tailcall puts "tailcalled puts"
    puts "demo1 exit"
}
proc demo {} {
    puts "demo enter"
    demo1
    puts "demo exit"
}
```

Here is the output when we invoke `demo`.

```
% demo
→ demo enter
  demo1 enter
  tailcalled puts
  demo exit
```

The `tailcall` arranges for the `puts` command to be invoked after the completion of the current call frame, which is that of `demo1`. It then forces the caller, again `demo1`, to **immediately complete** at which point the `puts` command set up by the `tailcall` is invoked. The `puts "demo1 exit"` line never gets executed.

This is all fairly straightforward. Now for the trickier example we mentioned. Let us rewrite `demo1` as follows.

```
proc demo1 {} {
    puts "demo1 enter"
    uplevel 1 {
        tailcall puts "tailcalled puts"
    }
    puts "demo1 exit"
}
```

Now the output below is somewhat puzzling (maybe) for a couple of reasons. First, you might expect that the `demo exit` line would not be printed as the `tailcall` is executed in the context of `demo` due to its being wrapped by the `uplevel`. Even stranger is that, unlike the previous example, the `tailcalled puts` is printed **after** `demo exit`.

```
% demo
→ demo enter
  demo1 enter
  demo exit
  tailcalled puts
```

Because it is run via `uplevel`, the `tailcall` runs in the call frame for `demo`, not that of `demo1`. Thus it schedules its argument to run after completion of `demo` and not `demo1`. Then it forces its caller to complete immediately with the `return` return code. The caller here is `uplevel` which propagates the `return` code causing `demo1` to immediately return. Evaluation of `demo` then continues as normal resulting in the `demo exiting` line being printed and finally when `demo` completes, the command set up by the `tailcall` gets to run.

Rarely will you need this level of minutiae but there you have it.

14.1.8. Hidden frames: info frame

```
info frame ?FRAMENUMBER?
```

We learnt about call frames in Section 14.1.2 and how the `info level` command provides access to the different call frames in the call stack. There are in fact a few hidden frames that are not visible via `info level`. These are hidden because they do not introduce new local variable scopes and as such do not have much programming significance. They contain metainformation about the script being executed, such as the method by which it is being executed (through `eval`, procedure calls, etc.), the source file it was defined in and so on.

The `info frame` command provides access to this information. If *`FRAMENUMBER`* is not provided, the command returns the frame level for the command. Otherwise, it returns a dictionary containing the metainformation for the frame at that level. If *`FRAMENUMBER`* is positive, it specifies the absolute frame level; if negative, it is relative to the current frame.

First, let us contrast `info level` and `info frame`.

```
proc demo {} {
    puts "demo context: level=[info level], frame=[info frame]"
    eval {
        puts "eval context: level=[info level], frame=[info frame]"
    }
    uplevel 1 {
        puts "uplevel context: level=[info level], frame=[info frame]"
    }
}
puts "global context: level=[info level], frame=[info frame]"
demo
→ global context: level=0, frame=1
  demo context: level=1, frame=2
  eval context: level=1, frame=3
  uplevel context: level=0, frame=3
```

If we look at the output, notice that

- In the global context, `info level` returns `0`, `info frame` returns `1`.
- The procedure call to `demo` increments both.
- The `eval` command increments only the `info frame` value as it does not add a call frame with a new variable scope.
- The `uplevel` command increments the `info frame` value but **decrements** the `info level` as we described in Section 14.1.5.

Other commands that evaluate scripts but do not introduce a new local variable scope, such as `source`, `try`, `if`, `while` etc. also behave in a manner similar to `eval`.

Let us now look at the information returned by the command. Write the following script to a file and then use `source` (Section 3.14) to evaluate it.

```
proc demo {} {
    demo2 "argument"
}
proc demo2 {arg} {
    puts "Frame: [info frame]"
    print_dict [info frame 2]
}
```

Then running `demo` will show the following output.

```
% demo
→ Frame: 3
  cmd     = demo2 "argument"
  file    = D:/Temp/TCL17486.TMP
  level   = 1
  line    = 2
  proc    = ::demo
  type    = source
```

The `cmd` element of the returned dictionary is the command being executed in that frame. We have introspected the frame one level up from where the `info frame` call is made. Thus the `cmd` entry shows the call to the `demo2` procedure.

The `type` element of the returned dictionary is `source` indicating that the command was defined inside a sourced file. It may also be `proc` for dynamically created procedure bodies, `eval`, `uplevel`, `try` etc. if located within a script being evaluated by those commands, or `precompiled` indicating it is a pre-compiled script.

The other entries of the dictionary are dependent on the value of the `type` element. In our example, these entries are

- `file`, containing the path to the file containing the command definition
- `line`, the line number within the file
- `proc`, the name of the procedure within whose body the command was invoked
- `level`, corresponding to the `info level` command.

See the reference pages for information about the dictionary elements for other values of `type` as well as use of the `interp debug` command to enable more detailed and exact reporting at the cost of performance. We do not go into further detail because of the limited utility of the `info frame` command. Its primary purpose is for building debuggers and similar tools. Unlike `info level`, you will seldom find it present in Tcl code.

14.2. Traces

Like manipulation of call frames, another facility in Tcl that is not commonly found in other languages is the ability to have program actions like variable access or command invocation trigger the execution of code (in addition to the command being invoked of course!). This capability, which we call *tracing*, can be used with great effect in a wide variety of scenarios:

- Implementation of read-only variables, validators, and the like, or modifying the behavior of commands without changing their implementation.
- A data flow style of programming where data modifications are propagated to other parts of the application. Spreadsheets are an example of this. So also user interfaces where programmatic updates and updates from the user are propagated in both directions.
- Development tools like profiles and debuggers that by their very nature need to be able "hook" into program and data flow to track and display changes.
- Resource cleanup in situations that cannot be handled with the normal Tcl exception handling capabilities.

The `trace` command implements this facility.

14.2.1. Tracing variables: trace add variable

```
trace add variable VARNAME OPS CALLBACK
```

We will start by exploring traces for variables. The `trace add variable` command turns on tracing of a variable.

The *`VARNAME`* argument specifies the name of the variable to be traced. Array variables as well as individual array elements are also supported.

The *`OPS`* argument must be a list of the operations shown in Table 14.1. These control the types of variable access that will trigger the trace.

`CALLBACK` is a command prefix (Section 14.3.1) to be invoked when the variable undergoes one of these operations. When a trace is triggered, the callback is passed three additional arguments. The first is the name of the variable on which the trace triggered. The second is the key of the element being accessed if the variable is an array and an empty string otherwise. The third is an operation value from Table 14.1.

There is no restriction on the number of traces you may add to a variable. If multiple traces are created, they will all be invoked unless one of them raises an exception. The order of invocation is the reverse order of their creation.

Table 14.1. Trace operations on variables

Operation	Description
`array`	Triggered on variable read or write via the `array` (Section 3.6.8) command.
`read`	Triggered just **before** the variable is read. The variable need not exist and can be set by the trace callback.
`unset`	Triggered when the variable is unset, either explicitly or implicitly when its containing scope is exited.
`write`	Triggered just **after** the variable is written. The trace callback can change the variable.

Let us start with a basic example to illustrate some finer points of traces. First some set up. We need a procedure that we will use as the callback for the traces.

```
proc tracer {varname elemname op} {
    puts "Trace: $op operation on variable $varname"
}
```

Now for the actual trace itself.

```
% trace add variable myvar {read write unset} tracer
```

Note from the above that we can set up a trace on a variable that does not even exist yet.

We can now try the various operations on the variable.

```
% info exists myvar
→ Trace: read operation on variable myvar
  0
% set myvar "foo"
→ Trace: write operation on variable myvar
  foo
% set myvar
→ Trace: read operation on variable myvar
  foo
% unset myvar
→ Trace: unset operation on variable myvar
% set myvar "bar"
→ bar
```

We can see our `tracer` procedure is triggered as expected in each case. A couple of points to note from the above:

- We can set up trace on a variable even before it is created.
- Existence checks with `info exists` also trigger a `read` trace.
- Once the variable is unset, the trace is also deleted so when we write to a variable of that name again, there is no trace in place.

The variable name passed to the trace callback is not always that on which the trace was applied. It is the name used to access the variable in the caller's context.

```
% trace add variable myvar {read write unset} tracer
% upvar 0 myvar linked_var
% set linked_var foo
→ Trace: write operation on variable linked_var
  foo
```

The callback sees the variable name `linked_var`, and not `myvar`. In most cases, such as accessing the variable with `upvar` as in the examples later, this is good enough. If you need the actual name on which the trace was applied, pass the name to the callback as an additional argument at the time the `trace` command is called.

Let us make our trace callback a little more sophisticated, reversing the variable content on reads and upper-casing on writes.

```
proc tracer {varname elemname op} {
    upvar 1 $varname var
    switch $op {
        read {
            set var [string reverse $var]
        }
        write {
            set var [string toupper $var]
        }
        unset {
            puts "Trace: \[info exists $varname\]=[info exists var]"
        }
    }
    return "This result of the callback is ignored"
}
```

The following points about trace callbacks should be noted in our example:

- Trace callbacks are invoked within the same context as the command that operates on the variable. Since our trace prefix is itself a procedure, it adds a call frame and `upvar` is needed to access the variable in the caller's context.
- Both `read` and `write` traces can modify the variable. The new value of the variable is what will be returned by the traced operation.

A legitimate concern is whether variable traces will fire recursively when we modify the variable within our callback. The answer is that Tcl is smart enough to disable read and write traces on a variable while a read or write callback is in progress. **However, note that this is not so (by design) if the callback is an unset operation.** Any `unset` traces will be triggered as usual. Moreover, traces are not disabled if the callback itself is in response to an `unset` operation.

Let us try this new version. A bit of imagination suffices to see how you can drive a colleague bananas without even touching their code.

```
% trace add variable myvar {read write unset} tracer
% set myvar "foo"
→ FOO
% puts $myvar
→ OOF
% puts $myvar
→ FOO
% unset myvar
→ Trace: [info exists myvar]=0
```

The output leads to some additional points of note:

- The write trace is called **after** the operation sets the new value of the variable. Our callback then updates it with the upper case version.
- The unset trace is also called **after** the variable is unset.
- The result of the original command reflects the new value of the variable **after** it is updated by the trace handler.
- The result of the trace callback `tracer` does not show up anywhere. It is ignored. However raised errors are treated differently as described below.

If a read or write callback raises an exception, the original command completes with the same exception. Exceptions raised during unset callbacks are ignored.

One non-obvious point with variable `unset` traces is that the command creates the specified variable if it is not already created. This means you can create traces on non-existent variables and have them fire as a means to detect when a variable scope is deleted. For example,

```
% proc demo {} {
    trace add variable NOSUCHVAR unset print_args
}
% demo
→ Args: NOSUCHVAR, , unset
```

Notice our trace fired when the `demo` procedure returned. Section 24.4.1 describes an application of this feature.

14.2.1.1. Tracing array variables

There are some special cases to be considered for tracing of array variables. Let us modify our tracing callback to just print its arguments.

```
proc tracer {varname elem op} {
    puts "Trace: varname=\"$varname\", elem=\"$elem\", op=\"$op\""
}
```

Tracing a specific element of an array is just like tracing a variable except that the name of the element is supplied as the second argument to the callback. The trace fires irrespective of whether the element is individually operated on or is part of an operation on the entire array.

```
% trace add variable arr(x) {read write unset} tracer
% set arr(x) 100
→ Trace: varname="arr", elem="x", op="write"
  100
% array set arr {x 0 y 1}
→ Trace: varname="arr", elem="x", op="write"
% unset arr
→ Trace: varname="arr", elem="x", op="unset"
```

If you want to track changes to **arbitrary** elements of an array as well as the array as a whole, specify the name of the array as the variable name and `array` as the operation along with other operations of interest.

```
% array set arr {x 0 y 1}
% trace add variable arr {array} tracer
% array get arr
→ Trace: varname="arr", elem="", op="array"
  x 0 y 1
% array set arr {a1 2 a2 3}
→ Trace: varname="arr", elem="", op="array"
% array unset arr a*
→ Trace: varname="arr", elem="", op="array"
```

When the `array` callback is made, it applies to the whole array and hence the second argument of the callback above, the element, is set to the empty string. The `array` callback does not indicate the type of operation, `get`, `set` etc.

If only `array` is specified as the operation as above, commands such as `array set` and `array get` will trigger the trace. However, reads and writes of individual elements will **not**:

```
% set arr(x)
→ 0
% set arr(z) 2
→ 2
```

For individual element access to be traced, we have to include the `read` and `write` operations. Now both array and element traces are invoked, the former triggered first:

```
% trace add variable arr {read write unset} tracer
% set arr(x)
→ Trace: varname="arr", elem="x", op="read"
  0
% array get arr
→ Trace: varname="arr", elem="", op="array"
  Trace: varname="arr", elem="x", op="read"
...Additional lines omitted...
```

Unsetting the entire array with `unset` does **not** trigger traces for **individual** elements unlike `array unset`.

```
% array unset arr z*
→ Trace: varname="arr", elem="", op="array"
  Trace: varname="arr", elem="z", op="unset"
% unset arr ❶
→ Trace: varname="arr", elem="", op="unset"
```

❶ Does not trigger unset traces on remaining individual elements.

14.2.1.2. Applications of variable tracing

We now illustrate some applications of variable tracing.

Lazy initialization

The fact that variables need not exist when traces are registered as well as the fact that they can be modified by the traces allows us to do lazy initialization.

Consider our simple `sum` procedure that returns the sum of the first N natural numbers. We can use traces to allow the application to access these values as array elements. For example, we should be able to say

```
puts "Sum of 1:5 is $sums(5)."
```

Now clearly we cannot predict which numbers **might** be of interest and even if we did, it might computationally expensive to pre-fill the array with values that might only be **potentially** used. Both these issues are easily solved with lazy initialization.

First we create the empty array and attach a variable trace to it that does the computation.

```
array set sums {}
proc calculate_sum {varname elem op} {
    upvar 1 $varname var
    if {! [info exists var($elem)]} {
        set var($elem) [sum $elem]
    }
}
trace add variable sums read calculate_sum
```

Our trace procedure is called every time an array element is read and the computed values are cached.

```
puts $sums(5)    → 15
puts $sums(3)    → 6
array names sums → 5 3
```

As you can imagine, this technique can be put to use in other caching scenarios.

Constant variables for Tcl 8

Here is an example of defining a variable as "read-only" or a constant. We will define a `constant` command for the purpose. Tcl 9 already has a `const` command (Section 3.6.9) so this would only be useful for Tcl 8.6 and earlier.

```
constant VARNAME VALUE
```

Any attempt to modify the variable will raise an error while keeping the variable unchanged.

```
proc constant {varname value} {
    upvar $varname var
    trace add variable var write \
        [lambda {constval name element op} {
            upvar 1 $name var
            set var $constval ❶
            throw {CONST MODIFY} "Attempt to modify a constant."
        } $value]
}
```

❶ Restore original value since it would have already been modified.

The above code implements the trace callback as an anonymous procedure (Section 3.5.9.4). When the callback is invoked, the value of the variable would have already changed so the original constant is passed to the callback separately as its first parameter.

Now any attempt to modify a `constant` variable is rejected.

```
% constant e 2.71828
% set e 0
Ø can't set "e": Attempt to modify a constant.
% set e
→ 2.71828
```

Data flow programming

Our next example is an outline of how a data flow program structure might be implemented. The example is beyond simplistic but should give you a flavor for how you might use variable traces as the basis for such a system. There are several examples of such use in the Tcler's Wiki[2].

We will have a 2x2 spreadsheet with cells numbered `A1-A2..B1-B2` and store cell values in global variables of the same name. Spreadsheet formulas are then almost trivial to implement in declarative style. Suppose the user has defined cell contents of `B1` and `B2` to be

```
B1 = A1 + A2
B2 = B1**2
```

This would translate to the following code

[2] https://wiki.tcl-lang.org

```
proc getval cell {
    upvar #0 $cell var
    return [expr {[info exists var] ? $var : 0}]
}
proc updateB1 {args} {
    set ::B1 [expr {[getval ::A1] + [getval ::A2]}]
}
proc updateB2 {args} {
    set ::B2 [expr {[getval ::B1]**2}]
}

trace add variable A1 {write unset} updateB1
trace add variable A2 {write unset} updateB1
trace add variable B1 {write unset} updateB2
```

Now we can see how updates are automatically propagated between cells.

```
set A1 3 → 3
set A2 4 → 4
set B2   → 49
set A2 2 → 2
set B2   → 25
```


The above example demonstrates “push” traces where changes to a variable are propagated to its dependents when it is written to. In some cases, a “pull” model can be more convenient. Instead of adding write traces to `A1`, `A2` we could add a read trace to `B1` and `B2` instead. When this trace fired, the **current** values of `A1`, `A2` would be used to compute a new value for `B1` / `B2`.

Resource management using traces

Traces can be used to ensure release of allocated resources even in the presence of errors. This is an alternative to using the `try..finally` command for the same purpose.

```
proc close_callback {chan args} { close $chan }
proc demo {} {
    set chanA [open /tmp/tcl-book/myfile.txt]
    trace add variable chanA unset [list close_callback $chanA]
    set chanB [open /tmp/tcl-book/myfile.txt]
    trace add variable chanB unset [list close_callback $chanB]
    puts in-demo:[chan names] ❶
}
demo
puts post-demo:[chan names] ❷
→ in-demo:file24154de7c50 stderr stdout stdin file24155305480 rc14
  post-demo:stderr stdout stdin rc14
```

❶ Open channels within procedure.
❷ Open channels after procedure completes.

The author generally prefers the use of `try` over `trace` for reasons of clarity. There are however circumstances where the use of `try` to free resources is not viable because the `finally` clause never gets to run. Two such cases are

- Deletion of an entire namespace.
- Deletion of a coroutine with the `rename` command.

In such cases, variable traces come to the rescue. This is described in detail with respect to coroutines in Section 24.4.1. The technique described there may be used to deal with namespace deletion as well.

14.2.2. Tracing commands

Tracing facilities for commands fall into two categories:

- tracing the lifetime of a command definition
- tracing command execution

We describe these in turn.

14.2.2.1. Tracing command lifetimes: trace add command

```
trace add command NAME OPS CALLBACK
```

The `trace add command` command registers a callback to be invoked when the specified command is renamed or deleted.

The argument *`NAME`* is the name of the command to be traced. Unlike for variable traces, this command must already exist. The argument *`CALLBACK`* is a command prefix (Section 14.3.1) to be invoked when the command is renamed or deleted. The *`OPS`* argument must be a list of elements from Table 14.2.

Table 14.2. Trace operations on commands

Operation	Description
`rename`	Triggered when a command is renamed. Renaming a command to an empty string is treated as a deletion and does not trigger this trace.
`delete`	Triggered when the command is deleted either by renaming it to the empty string or by deletion of the containing namespace.

`CALLBACK` is invoked with three arguments: the fully qualified name of the command, the fully qualified new name or an empty string if the operation is `delete`, and `rename` or `delete` indicating the operation.

As for `read` and `write` traces on variables, command traces for a command are disabled when a trace callback for **that** command is in progress.

Command traces tend to be less common than variable traces but there are still situations where they are useful. One example is packages where commands are "proxies" for external objects, such COM on Windows. The package can ensure the external object is released at the appropriate time by placing a trace on the command deletion.

Here is a simple example illustrating command traces.

```
% namespace eval ns { proc demo {} {} }
% trace add command ns::demo {rename delete} tracer
% rename ns::demo demo2 ❶
→ Trace: varname="::ns::demo", elem="::demo2", op="rename"
% rename demo2 ""
→ Trace: varname="::demo2", elem="", op="delete"
```

❶ Note trace will remain attached after renaming.

Redefinition of a procedure is treated as a deletion and the trace fires accordingly. The new definition will **not** have the trace attached.

14.2.2.2. Tracing command execution: trace add execution

```
trace add execution NAME OPS CALLBACK
```

The `trace add execution` command is a very powerful tool that can provide insight into the exact sequence of commands executed in a program.

Powerful as they are, execution traces also extract a large performance penalty. Their use should therefore be limited to debugging and troubleshooting purposes and is not recommended as part of normal program flow.

The execution trace can track the invocation of a specified command or **all** command execution for the duration of that command.

`NAME` is the name of an existing command whose execution is to be traced and *`CALLBACK`* is a command prefix (Section 14.3.1) to be invoked when the trace is triggered. *`OPS`* must be a list of elements from Table 14.3.

Table 14.3. Trace operations on command execution

Operation	Description
`enter`	Triggered before the **specified** command begins execution.
`leave`	Triggered after the **specified** command completes normally or with an error exception.
`enterstep`	Triggered before **every** command during the execution of the specified command including nested calls to any depth.
`leavestep`	Triggered **after** the completion of every command during the execution of the specified command including nested calls.

When a trace is invoked for `enter` and `enterstep` triggers, *`CALLBACK`* is invoked with two additional arguments. The first is the full command string and the second is `enter` or `enterstep` depending on the trigger.

When invoked for `leave` and `leavestep` triggers, four arguments are added. The first is the command string, the second is the return code (Section 15.2.1) from the command invocation, the third is its result and the fourth is the trigger operation, `leave` or `leavestep`.

The scripts below show the difference between the tracing modes.

```
proc tracer args { puts "Trace: [join $args {, }]" }
proc demo {args} { demo2 X Y }
proc demo2 {args} { demo3 }
proc demo3 {} {return "result" }
trace add execution demo {enter leave} tracer
demo
→ result
  Trace: demo, enter
  Trace: demo, 0, result, leave
```

Notice our tracer logs the invocation and completion of the `demo` command but nothing in between. On the other hand, adding traces for `enterstep` and `leavestep` logs **all** commands while `demo` was executing.

```
trace add execution demo {enterstep leavestep} tracer
demo
→ result
  Trace: demo, enter
  Trace: demo2 X Y, enterstep
  Trace: demo3, enterstep
  Trace: return result, enterstep
  Trace: return result, 2, result, leavestep
  Trace: demo3, 0, result, leavestep
  Trace: demo2 X Y, 0, result, leavestep
  Trace: demo, 0, result, leave
```

Let us redefine `demo3` to raise an error instead.

```
proc demo3 {} {error "Something horrible happened."}
demo
Ø Something horrible happened.
  Trace: demo, enter
  Trace: demo2 X Y, enterstep
  Trace: demo3, enterstep
  Trace: error {Something horrible happened.}, enterstep
  Trace: error {Something horrible happened.}, 1, Something horrible happened.,...
  Trace: demo3, 1, Something horrible happened., leavestep
  Trace: demo2 X Y, 1, Something horrible happened., leavestep
  Trace: demo, 1, Something horrible happened., leave
```

Again, note the trace triggers on completion of each procedure even on exceptions where the return code is shown as `1` as opposed to `0` for a normal return.

Execution traces are not normally used because of their performance impact. Nevertheless, they can be indispensible for fault diagnosis, particularly in the field since they can be configured with no changes to the application source.

14.2.3. Deleting a trace: trace remove

```
trace remove variable NAME OPS CALLBACK
trace remove command NAME OPS CALLBACK
trace remove execution NAME OPS CALLBACK
```

All three forms of traces can be deleted with the `trace remove` command.

The *`NAME`* and *`OPS`* arguments have the same semantics as for the `trace add` command — they identify the variable or command and the operations for which the trace is to be deleted. Since there may be multiple traces on a variable or command, *`OPS`* and *`CALLBACK`* identify the specific trace to be removed. They must match the corresponding arguments that were used to initiate the trace.

```
trace remove execution demo {enterstep leavestep} tracer
demo
Ø Something horrible happened.
  Trace: demo, enter
  Trace: demo, 1, Something horrible happened., leave
```

Notice that only the specified triggers were removed. The `enter` and `leave` remained active.

To reiterate the point about both the *`OPS`* and *`CALLBACK`* arguments having to be the same as in the initiating trace command, suppose we only wanted to remove the `enterstep` trigger. The following would **not** work.

```
trace remove execution demo enterstep tracer
```

The *`OPS`* argument would not match the initiating `trace` so the trace removal would be silently ignored. For the desired effect you have to remove the complete trace as in the prior example and add a new one with just the `leavestep` trigger specified.

An attempt to remove a trace on a non-existent variable is silently ignored but an attempt to remove a trace on an undefined command will raise an exception.

14.2.4. Inspecting traces: trace info

```
trace info variable NAME
trace info command NAME
trace info execution NAME
```

The `trace info` command can be used to retrieve active traces. The result of the command is a list of pairs each of which contains the *`OPS`* and *`CALLBACK`* arguments that were supplied to the `trace add` command.

Let us check the execution traces on our `demo` procedure after adding back the trace that we previously removed for demonstration purposes.

```
trace add execution demo {enterstep leavestep} tracer
trace info execution demo
→ {{enterstep leavestep} tracer} {{enter leave} tracer}
```

We can use this information to remove all traces from a variable or command.

```
foreach trace [trace info execution demo] {
    trace remove execution demo {*}$trace
}
```

14.3. Code construction

In dynamic languages like Tcl, it is common for code fragments to be passed around, evaluated and even constructed on the fly. Examples include

- Operation of commands like `lsort` (Section 5.21) and `dict filter` (Section 6.17) can be customized through callbacks.
- Event handlers (Chapter 19) and `traces` (Section 14.2) use callbacks for notification purposes.
- Although it may not be obvious if you are coming from other languages, even the "body" of commands like `eval` (Section 3.13), `try` (Section 15.4.3), `if` (Section 3.7), and `while` (Section 3.9) are just arguments and not any special syntactic constructs. You can thus pass dynamically constructed code as their bodies.
- Metaprogramming, which we discuss in Section 14.4, is based on the ability to construct and execute code at runtime.

In this section, we provide some hints and tips related to these aspects of Tcl.

14.3.1. Scripts versus command prefixes

For starters, we need to distinguish between callbacks that are *scripts* versus arguments that are *command prefixes*. Commands that take script arguments evaluate them as Tcl scripts with (potentially) multiple commands and following the usual Tcl syntax and substitution rules. On the other hand, commands that take a command prefix argument treat the argument as a **single** command which is in a list form containing the command name and possibly some arguments to be passed to it. In both cases, when the callback is invoked additional arguments **may** be appended containing specific information about why it is being invoked.

The following mock procedures illustrate the difference. The `script_cb` procedure is written to accept a script callback as an argument while `cmd_cb` takes a command prefix. We will pass the same `callback` argument to both.

```
set callback {print_args A ; print_args B C}
proc script_cb {script} {
    uplevel 1 $script "(script)"
}
proc cmd_cb {cmdprefix} {
    tailcall {*}$cmdprefix "(command)"
}
```

Notice the difference in the output below in the two cases.

```
% script_cb $callback
→ Args: A
  Args: B, C, (script)
% cmd_cb $callback
→ Args: A, ;, print_args, B, C, (command)
```

This difference arises because the `script_cb` command executes the callback as a script where the `;` character is treated as a command separator. In the case of `cmd_cb` it is treated simply as an argument.

Both forms of callback arguments are commonly seen and you have to just be aware of what type of callback is expected by a command.

14.3.1.1. Constructing command prefixes

In the example above, we passed a (brace enclosed) string as an argument for the purposes of contrasting command prefixes with scripts. However, the recommended way to pass a command prefix as a callback is by constructing it as a list. For example,

```
% set some_value "First arg"
→ First arg
% cmd_cb [list print_args $some_value ";" "Third arg"]
→ Args: First arg, ;, Third arg, (command)
```

Providing the callback as an interpolated string would require more care, such as escaping whitespace and special characters, to ensure it is parsed correctly as a list of arguments.

It is also common to use an anonymous procedure as a command prefix instead of defining a named procedure for a one time use. For example, the custom sorting example from Section 5.21 can be written as

```
set part_numbers {part_100_b PART_100_C PART_20_B}
lsort -command [lambda {s1 s2} {
    return [expr {[string length $s1] - [string length $s2]}]
}] $part_numbers
→ PART_20_B part_100_b PART_100_C
```

where we have used the `lambda` utility procedure from Section 3.5.9.4 to define an anonymous procedure for comparing strings in order of their length.

14.3.1.2. Constructing scripts

Constructing scripts is more involved than command prefixes because while the latter have a limited structured form, scripts can be full blown Tcl programs. In their simplest form, scripts are enclosed in braces as a literal string. We have seen this frequently in the definition of a procedure body, `if` statement and so on. This is not particularly useful in dynamic script construction though, because in most cases the script is at least partly built from runtime information and not completely known at the time it is written. Enclosing the script in braces precludes use of variable and command substitutions in the generation of the script.

There are several alternatives for building scripts by combining "static" fragments with dynamic ones at runtime:

- The script can be composed from a sequence of commands that append literals and variable fragments. This is the most flexible alternative but suffers from a lack of readability where the structure of the generated script is not readily apparent.
- Alternatively, the script can be constructed as a literal string in double quotes instead of braces. The variable parts of the scripts can then simply be variable references or bracketed commands that are replaced through the normal string interpolation rules. This is a reasonable approach when the constructed script is simple. For even moderately complex scripts however, several issues arise. Variable references and bracketed commands that are part of the *generated* script need to be escaped so they are not substituted at script generation time. This escaping of special characters and newlines can become tricky. There is also a loss of readability as in the previous alternative.
- For more complex scripts, it is often easiest to write the script as a template with "place holders" for the dynamic parts. These are then replaced at runtime through commands such as `subst` (Section 4.19), `format` (Section 4.20) and `string map` (Section 4.10).

This last method is illustrated in Section 14.4.1.

14.3.2. Capturing namespace contexts in callbacks

One consideration that arises when constructing scripts passed as callbacks is definition of the namespace in which the callback should execute. Most commands such as `after` execute scripts in the global context. To execute the callback in another context, some additional steps are needed. Since we have not discussed namespaces yet, we will postpone a discussion of this topic to Section 16.2.1.

14.4. Metaprogramming

What is *metaprogramming*? Roughly speaking, metaprogramming involves writing a program **that in turn writes a program** to do the desired task. In some cases metaprogramming makes for simpler code while in others it optimizes performance by generating specialized code at runtime. Tcl lends itself naturally to this style of programming as we have already seen with some simple examples involving procedure redefinitions and such. We will now present some additional illustrations of metaprogramming. We will see another example in Section 23.12.2.

14.4.1. Procedures with initializers

In Section 3.5.5 we saw how a procedure could redefine itself to do one-time initialization. That required some boilerplate code to be written for every procedure that wanted to do this. This boilerplate followed the pattern

```
proc NAME {ARGLIST} {
    INITCODE
    proc NAME {ARGLIST} {
        BODY
    }
    tailcall {*}[info level 0]
}
```

We can generalize this by introducing an enhanced form of the `proc` command, which we will imaginatively call `proc_ex`, that will generate this boilerplate for us. Our new command takes an additional argument which is the initialization script.

```
proc_ex PROCNAME ARGS INIT BODY
```

The command will create a new procedure called *`PROCNAME`* just as the `proc` command does except that the **first** time it is called, the created procedure will run the *`INIT`* script before running *`BODY`*.

We can implement `proc_ex` by just using the above pattern as a template for defining the target procedure and substituting for the variable parts in the template. For example, the following implementation uses `string map` (Section 4.10) to do the needful.

```
proc proc_ex {name arglist initcode body} {
    if {![string match ::* $name]} {
        set ns [uplevel 1 {namespace current}]
        set name ${ns}::$name
    }

    set template {
        proc NAME {ARGS} {
            INIT
            proc NAME { ARGS } { BODY }
            tailcall {*}[info level 0]
        }
    }
    set replacements [list NAME $name ARGS $arglist INIT $initcode BODY $body]
    eval [string map $replacements $template]
}
```

The first part of the procedure merely ensures the name of the procedure to be defined is appropriately qualified irrespective of the namespace context of the caller. In the second part, we take the generalized procedure template we laid out above, replace the variable parts of the template with the actual values using string map, and then execute the generated procedure definition.

To understand how it works, we can introspect the generated code for an example.

```
proc_ex say_hello {message} {
    puts "Loading package msgcat"
    package require msgcat
} {
    puts [msgcat::mc $message]
}
```

The above defines a `say_hello` procedures whose implementation we can examine with `info body`.

```
% info body say_hello
→

       puts "Loading package msgcat"
       package require msgcat

               proc ::::say_hello { message } {
       puts [msgcat::mc $message]
   }
               tailcall {*}[info level 0]
```

The formatting of the generated code is a bit of a mess as we have not bothered to prettify it. The implementation of `say_hello` (generated by `proc_ex`) runs the initialization code and then redefines itself with the main procedure body. It finishes by calling this redefined version of itself.

Let us call our procedure for the first time.

```
% say_hello "Hello World!"
→ Loading package msgcat
  Hello World!
```

As you can see the initialization code is executed before the main body. Moreover, if we examine the body of the procedure, we find it has changed.

```
% info body say_hello
→
       puts [msgcat::mc $message]
```

And naturally, when we invoke it a second time, there is no attempt to load the `msgcat` package.

```
% say_hello "Hello again!"
→ Hello again!
```

To recap the benefits of our `proc_ex` command,

- We can postpone any expensive initialization (loading `msgcat` in our example) until the time it is actually needed.
- Subsequent calls after the first are streamlined as they neither attempt to load the package nor even have to check for the same.
- Because we have wrapped this one-time initialization within our `proc_ex` procedure, it is simple to use. A procedure that requires one-time initialization does not need to reinvent the wheel.

The `string map` (Section 4.10) command is only one way of generating a script from a template. You could also use commands like `subst` (Section 4.19) or `format` (Section 4.20). An implementation of `proc_ex` that uses `format` is shown below.

```
proc proc_ex {name arglist initcode body} {
    if {![string match ::* $name]} {
        set ns [uplevel 1 {namespace current}]
        set name ${ns}::$name
    }
    eval [format {
        proc %1$s { %2$s } {
            %3$s
            proc %1$s { %2$s } { %4$s }
            tailcall {*}[info level 0]
        }
    } $name $arglist $initcode $body]
}
```

14.4.2. Parsing data

A common task in programming is parsing of structured data or text. One technique that you will see in the Tcl world is to transform the data into a Tcl script that embeds Tcl commands within the data and then execute the generated script.

This is easiest explained through an example. We will use the following Tcler's Wiki[3] code derived from Stephen Uhler's famous 4-line HTML parser.

```
proc html_parse {html callback} {
    set re {<(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>}
    set sub "\}\n[list $callback] {\\2} {\\1} {\\3} \{"
    regsub -all $re [string map {\{ \&ob; \} \&cb;} $html] $sub script
    eval "$callback PARSE {} {} \{ $script \}; $callback PARSE / {} {}"
}
```

The intent is to transform the HTML text to a Tcl script where each HTML tag results in the invocation of a command which is passed the tag as a parameter. To understand what `html_parse` is doing, let us **interactively** execute a slightly simplified version of the above implementation line by line.

We start off by defining variables corresponding to the arguments passed to `html_parse`. These will serve as the "arguments" to our interactive execution.

[3] https://wiki.tcl-lang.org

```
set html {
    <p class='important'>Something <b>really</b> important.</p>
    <p>A second paragraph</p>
}
set callback html_cb
```

The first line of `html_parse` defines a regular expression that matches opening as well as closing HTML tags.

```
% set re {<(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>}
→ <(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>
```

The second line of `html_parse` defines the substitution used with `regsub` to map each tag to a call to the callback command.

```
% set sub "\}\n[list $callback] {\\2} {\\1} {\\3} \{"
→ }
  html_cb {\2} {\1} {\3} {
```

Our intent is that a tag of the form `<body>` will result in a call to `html_cb` (the callback procedure passed in) with the `<body>` passed as an argument along with the succeeding text. We will see this in a minute.

The third line calls the `regsub` command (Section 10.2) to transform the HTML text into an equivalent Tcl script fragment.

```
% regsub -all $re $html $sub script
→ 6
% puts $script
→
      }
  html_cb {p} {} {class='important'} {Something }
  html_cb {b} {} {} {really}
  html_cb {b} {/} {} { important.}
...Additional lines omitted...
```

Note the output is a script fragment, not an entire script (hence the leading and trailing brace characters). It calls the specified command passing four parameters. The complete script that is passed to `eval` on the fourth line of `html_parse` would then look like the output of the `puts` command below.

```
% puts "$callback PARSE {} {} \{ $script \}; $callback PARSE / {} {}"
→ html_cb PARSE {} {} {
      }
  html_cb {p} {} {class='important'} {Something }
  html_cb {b} {} {} {really}
  html_cb {b} {/} {} { important.}
...Additional lines omitted...
```

Thus invoking our 4-line HTML parser as follows

```
html_parse $html html_cb
```

will result in the script printed above being generated and evaluated. We can now gain a better understanding of how the script works. It transforms the passed HTML text such that each HTML begin and end tag is converted to a call to the passed callback command, `html_cb` in our example, with four arguments:

- the name of the tag, such as `p` or `b`,
- an argument that is empty if it is the beginning of the tag and `/` if it corresponds to the tag termination,
- any attributes for the tag,
- the text content until the start of the next tag.

The script uses a special tag name `PARSE` that allows the callback command to recognize the start and end of parsing for any required state initialization or finalization.

All we need to do now is define our callback command `html_cb` to do the desired parsing action. Let us define a trivial transform to remove all attributes and convert bold tags to italics.

```
proc html_cb {tag place attrs content} {
    if {$tag ne "PARSE"} {
       if {$tag eq "b"} {set tag "I"}
       puts -nonewline "<$place$tag>$content"
    }
}
```

Now examine the output when our sample HTML fragment is processed as below.

```
% puts $html
→
      <p class='important'>Something <b>really</b> important.</p>
      <p>A second paragraph</p>

% html_parse $html html_cb
→ <p>Something <I>really</I> important.</p>
      <p>A second paragraph</p>
```

This technique of transforming data to a script has potential security issues when the data comes from an unknown (and potentially malicious) source. Although this can be guarded against with proper escaping and quoting of the input data, it is advisable to execute the generated script in a safe interpreter (Section 23.10).

Our HTML parser is simplistic and does not take into account all of HTML syntax details and idiosyncracies. It is meant to illustrate the “data to script” transform technique. Several HTML parsing libraries are available as third party packages.

14.4.3. Code generalization

Consider writing a procedure that given a pair of lists, returns a list of all possible pairs containing an element from each list.

```
proc pairs {la lb} {
    set res {}
    foreach a $la {
        foreach b $lb {
            lappend res [list $a $b]
        }
    }
    return $res
}
pairs {a b} {1 2 3}
→ {a 1} {a 2} {a 3} {b 1} {b 2} {b 3}
```

What if we wanted to generate triples from three lists instead? That would be easy — we just add another nested loop. But what if we wanted to generalize the procedure to be able to take an **arbitary** number of list arguments? And not limit the generalization to just generating list pairs? This metaprogramming example, adapted from the Wiki[4], is one such generalization.

The syntax of the `forall` command that we will write is very similar to that of the built-in `foreach` command. The difference is that `forall` will process the lists in **nested** fashion while `foreach` processes them in parallel in the same iteration.

```
forall VAR LIST ?VAR LIST …?  BODY
```

The implementation constructs a script containing `foreach` loops nested to the required depth with the innermost loop containing the *`BODY`* script to be executed. The constructed script is then evaluated in the caller's scope.

```
proc forall args {
    if {[llength $args] < 3 || [llength $args] % 2 == 0} {
        return -code error "wrong \# args: should be \"forall varList list \
            ?varList list ...? body\""
    }
    set body [lindex $args end]
    set args [lrange $args 0 end-1]
    while {[llength $args]} {
        set varName [lindex $args end-1]
        set list    [lindex $args end]
        set args    [lrange $args 0 end-2]
        set body    [list foreach $varName $list $body]
    }
    uplevel 1 $body
}
```

We can emulate out `pairs` command as below.

[4] http://wiki.tcl-lang.org/2546

```
% forall x {a b} y {1 2 3} { lappend res [list $x $y] }
% set res
→ {a 1} {a 2} {a 3} {b 1} {b 2} {b 3}
```

But now, with this generalized procedure, we are not limited to producing pairs. We can do something more creative, like producing strings instead! And this time with a different number of lists.

```
% set res ""
% forall x {a b} y {1 2 3} z {M N} { append res $x$y$z }
% set res
→ a1Ma1Na2Ma2Na3Ma3Nb1Mb1Nb2Mb2Nb3Mb3N
```

Finally, here is the generalization of our `pairs` procedure. Instead of producing pairs, it will produce tuples composed of elements from an arbitrary number of lists. The procedure first constructs the list of `forall` arguments consisting of alternating loop variable names and list values. It then calls `forall` to do the hard work of generating combinations.

```
proc tuples args {
    set res {}
    set listargs {}
    set body "lappend res \[list"
    foreach arg $args {
        set loopvar v[incr i]
        append body " \$$loopvar"
        lappend listargs $loopvar $arg
    }
    append body "\]"
    forall {*}$listargs $body
    return $res
}
```

And to prove it works as desired,

```
% tuples {1 2} {a b c}
→ {1 a} {1 b} {1 c} {2 a} {2 b} {2 c}
% tuples {1 2} {a b c} {X Y}
→ {1 a X} {1 a Y} {1 b X} {1 b Y} {1 c X} {1 c Y} {2 a X} {2 a Y} {2 b X} {2 b ...
```

Now, for this specific problem, there are easier ways to code directly without metaprogramming but what the `forall` procedure has done for us is **to make it easy to iterate over lists in nested fashion in a very generalized way without having to write custom code every time**.

14.5. Command history: history

When running in interactive mode, Tcl keeps a history of all commands entered by the user. These commands can be recalled or manipulated with the `history` ensemble command.

The `history` commands described in this section are rarely directly used by applications. Their primary use is in writing a custom shell, similar to `tclsh` (Section 2.2.2), `wish` (Section 2.2.3) or `tkcon` (Section 2.2.4), that accepts input commands from the user and provides command history and recall similar to those shells. You can therefore comfortably skip this section.

If no arguments are passed, `history` returns a human-readable list of previous commands.

```
% puts "This is the first command"
→ This is the first command
% set i 1
→ 1
% incr i
→ 2
% history
→      1  puts "This is the first command"
       2  set i 1
       3  incr i
       4  history
```

The above `history` command is just a short form of the `history info` command which allows you to optionally specify the number of entries to be returned.

```
% history info 2
→      4  history
       5  history info 2
```

Each command has a sequence number associated with it. The `history nextid` command returns the sequence number of the **next** entry.

```
% history nextid
→ 7
```

Entries in the command history are referred to as command history events (not to be confused with events as described in Chapter 19). They can be referenced in multiple ways:

- A positive integer is the sequence number of an entry in the history.
- A negative integer is interpreted relative to the current sequence number.
- Any other string is first searched for backward as an exact prefix of an entry in the history and if not found, is searched as a glob pattern.

We can use these forms with any `history` subcommand that operates on individual entries. For example, the `history event` command returns the corresponding entry from the history.

```
% history event ❶
→ history nextid
% history event 2
→ set i 1
```

❶ By default the previous entry in the history is returned

An entry in the command history can be re-executed with the `history redo` command. Again, you can use any of the forms above for referencing.

```
% history redo -8 ❶
→ This is the first command
% history redo 2 ❷
→ 1
% history redo incr ❸
→ 2
% history redo ❹
→ 3
```

❶ Execute command 8 entries back
❷ Execute command with sequence number 2
❸ Execute command matching `incr`
❹ Repeat previous command

It is also possible to modify the command history in various ways. The `history add` command adds a command to the history and optionally executes it if the `exec` argument is specified.

```
% history add [list puts "This will not print"] ❶
% history add [list puts "This will print"] exec ❷
→ This will print
```

❶ Command is not evaluated
❷ Command is evaluated

The command `history change` on the other hand modifies an existing command.

```
% history event 15
→ history add [list puts "This will not print"]
% history change [list puts "This will now also print"] 15
→ puts {This will now also print}
% history event 15
→ puts {This will now also print}
```

The entire command history can be erased by calling `history clear`. Further commands added to the history will begin with sequence number 1.

```
% history clear
% puts "History never repeats itself!"
→ History never repeats itself!
% history
→      1  puts "History never repeats itself!"
       2  history
```

The `history keep` command retrieves or sets the maximum number of entries that are maintained in the history.

```
% history keep
→ 20
% history keep 100
→ 100
```

14.6. Counting command invocations: info cmdcount

The `info cmdcount` command returns the total number of commands invoked in a Tcl interpreter since it was started.

```
info cmdcount → 417604
info cmdcount → 417605
```

The count of command invocations is seen in two scenarios. One involves the use of safe interpreters where a limit is set for the number of commands an interpreter is allowed to execute before it is terminated. We will examine this in Chapter 23.

The other use of `info cmdcount` is to generate identifiers at run time that are unique **within that interpreter**. Examples include naming of objects, handles for resources, coroutines and so on.

```
proc make_id {{prefix id}} { return ${prefix}[info cmdcount] }
```

We can use this to generate new unique identifiers.

```
make_id      → id417609
make_id coro → coro417612
```

Of course, it is as simple to maintain a counter in lieu of using `info cmdcount` for this purpose.

The command invocation count is not incremented in certain cases due to optimizations in the Tcl byte code compiler. However, it is safe to use for the above purpose as the `info cmdcount` command itself will increment the count.

15

Errors and Exceptions

> An error does not become truth by reason of multiplied propagation, nor does truth become error because nobody sees it.
>
> — Mahatma Gandhi

Dealing with unexpected failures is an important part of any non-trivial program. Failures may result from programming errors such as accessing undefined variables, user actions such as attempts to open a protected file, hardware conditions etc. In this chapter we examine Tcl's facilities for handling errors and special conditions.

15.1. Dealing with failures

When such a failure or error occurs, software may deal with it in any of the following ways:

1. Ignore the issue. Stout denial, as Wodehouse would say, that a problem could possibly exist. We will pretend we never do this.
2. Terminate the program. This is a fair strategy in some circumstances. If a file copy program does not have access to the target directory, it is perfectly reasonable for it to simply exit with an appropriate message.
3. Report the error through a special result value that could not possibly be a valid result. For example, a command that returns a length could return `-1` to signal an error.
4. The command may explicitly pass back a status code in addition to the command result, either through an additional output parameter or by returning the result and status as a pair.
5. One last alternative is the subject of this chapter — *exceptions*. When an error or failure is detected, the code detecting the error *throws* or *raises* an exception. This causes the normal flow of execution to be aborted and control is passed back up the call stack until an *exception handler* is found that is defined for that exception. This exception handler is expected to take the appropriate actions to deal with the error condition.

Exceptions are the preferred mechanism for error reporting for several reasons:

- We really should not be considering the first alternative.
- Alternative 2 is a viable alternative only at the top application level.
- Alternative 3 is only possible if some result values cannot be logically valid.
- Alternative 4 makes for awkard syntax to use the result.
- Alternatives 3 and 4 both require an explicit check for errors, often skipped by programmers in a hurry.

On the other hand,

- Exceptions do not clutter up the main logic with explicit error checks, making it easier to read and reason about the program.
- Exceptions make it easy to handle errors at an appropriate point in the call hierarchy, which is not necessarily at the point the error is discovered.

In addition to error handling, the exception mechanism has other uses in Tcl as well, such as implementation of custom control structures that are on par with the built-in ones like `for` or `while`.

We will start our discussion of error handling and exceptions by describing the underlying return code mechanism on which they are based.

15.2. Return codes and the option dictionary

So far we have been happily working under the assumption that Tcl commands return a single result value (though the value itself may be a collection of multiple values). In reality, every command completion actually produces *three* values, the command *result*, an integer *return code*, and a *return options dictionary*.

The command result is the result value from invocation of a command that we have been making use of all along. Additionally, a command completion also has an associated return code and return options dictionary. The return code can be roughly thought of as a status. The return options dictionary contains additional information that is usually relevant only in the case of errors. We will now look at how these are set, retrieved and the associated semantics.

15.2.1. Return codes

Tcl executes a script, including procedures, `eval` arguments, bodies of `if` and `while` statements etc., as a sequence of commands. Each command completes with a result, a return code and a return options dictionary. A return code of `0` signifies normal completion of a command and Tcl continues execution with the next command in the script. For any other value of the return code, execution of the script stops and the result and return code from the last executed command becomes the result and return code of the script.

What happens next depends on the caller of the script. The caller may be the Tcl procedure evaluation code, a built-in command, looping or conditional commands like `while` and `if`, or even user-defined. This caller may choose to take a specific action for certain return codes. Other codes that it chooses not to handle are then passed further up the call stack where they are handled in the same manner.

Let us illustrate the use of return codes and associated semantics through an example — the `break` (Section 3.11) command used for early termination of loops. Like all other commands in Tcl, `break` is not a special keyword as in other languages. It is simply a command like any other and returns a result and a return code. The **result value** for the `break` command is always an empty string and the **return code** is always `3` (we will show this in a bit). In the case of `break`, both values are implicit though that is not the case for all commands. **Now, how this return code is dealt with by the calling command is entirely up to it**. Tcl itself does **not** mandate any semantics on a return code value of `3`.

The calling command may choose to take any action it pleases in response to the called command returning code `3`:

- When invoked within a looping construct like `while` (Section 3.9) or `foreach` (Section 5.8), the implementation of those commands treat the return code of `3` from the invocation of **any** command, not just `break`, as a signal to terminate the loop.
- In the Tk GUI extension, bindings for events such as mouse clicks allow multiple scripts to be registered. These are run sequentially on the occurence of the event. If any script returns a code of `3`, Tk treats this as a directive to skip the remaining registered scripts.
- Within a `catch` command (Section 15.4.1), a return code of `3` will result in the `catch` command returning `3` as its **result** (not its return code).
- Within the **outermost** level of a procedure body evaluation, any command returning a code of `3` will result in the procedure itself returning immediately with a return code of `1/error` (we will see what this means momentarily).

Thus each invoking command will treat a `break` command within its scope in a slightly different manner. The semantics are completely up to the command. Now naturally, the behaviour has to be documented by the command and similar commands should treat the same return code value in consistent fashion. It would be no good if the `foreach` command treated the return code `3` from `break` as a loop termination command while the `lmap` command treated it as a signal to skip one iteration!

The return code from a command invocation may be any integer value but Tcl defines five specific values, and associated mnemonics, shown in Table 15.1. Applications and packages may define their own custom return codes (Section 15.3.3).

Table 15.1. Tcl-defined return codes

Code	Mnemonic	Description
`0`	`ok`	The command completed normally.
`1`	`error`	Indicates an error condition.
`2`	`return`	Signals the caller that it should stop its own execution and return control back to the its own caller.
`3`	`break`	This return code expects callers to be looping constructs and signifies termination of the loop. As we saw in our introductory example, it may also be used in other situations where a sequence of commands is being executed to skip the remaining commands in the sequence. We stress again that this behaviour is **not** built into Tcl. It is dependent on how the command receiving the return code chooses to handle it.
`4`	`continue`	Like the `break` return code, this is used with looping constructs which are then expected to skip the remaining part of the current iteration and continue with the next one.

We will use the term *normal return* whenever a command completes with a return code `0/ok`. Any other return code value will be termed as an *exceptional return* or just *exception*. Note that an exception is not necessarily an error (e.g `break`).

15.2.2. Return code propagation

Let us now take a closer look at how these return codes are propagated and how they control the flow of execution in a Tcl program.

15.2.2.1. Propagating **break** and **continue** return codes

We will use the following simple `while` loop to demonstrate.

```
set i 0
while {1} {
    incr i
    if {$i == 1} {
        incr i
        continue
    }
    puts "i = $i"
    if {$i >= 4} break
}
→ i = 3
  i = 4
```

We will focus on the execution of the `while` body. In the first iteration of the loop,

- The command `incr i` is invoked. It completes normally with a **result** of `1` (the value of `i`) and a **return code** of `0` / `ok` that signals the normal completion.
- Because the command completed normally, the evaluation of the `while` body continues with the next command, the `if {$i == 1} …` statement.
- As its condition evaluates to `true`, the `if` statement begins executing its body. Here we are actually glossing over the fact that the condition evaluation itself involves return codes.
- The first statement in the `if` body is an `incr` which as before completes successfully. Its return code is therefore `0` / `ok` and evaluation moves on to next command in its body.
- This is now where things get more interesting. The `continue` command's sole purpose in life is to complete with a return code of `4` / `continue`. When a script evaluation is handed a exceptional return code (i.e. any other value than `0` / `ok`), instead of continuing with the next command in the script, it returns the same return code to its caller which in our example is the `if` command.
- The `if` command also does not know how to deal with a return code of `4` / `continue`, and it is therefore propagated up the call stack to the evaluation of the `while` body and then to the `while` command itself.
- The `while` command does incorporate special handling for the `break` and `continue` return codes. When it gets back a `continue` code here, it proceeds with the next iteration of its body. Note once again that this is the choice of the `while` command implementation, **not** Tcl.

The second iteration behaves in similar fashion:

- As before the `incr` command return code is `ok`.
- The condition for the `if` command is `false` so its body is not executed. The condition boolean is independent of the `if` command return code which is `ok`.
- The `puts` command also completes normally with a return code of `ok`.
- The second `if` statement condition is `true`. Its body is simply the `break` command which as we said completes with a return code of `break`. As we described for the `continue` command above, this is propagated up through the evaluation of the `if` body, the `if` command itself, the evaluation of the `while` body until it is passed to the `while`. At that point, the `while` command on receiving the `break` return code, reacts by terminating the loop iterations.

To summarize the above then,

- Every command evaluation returns a return code independent of the command result.
- If the return code is `ok` the caller proceeds as normal. It may also check for, and handle, one or more specific exception codes (like `break` and `continue` above).
- All codes that the caller does not handle are propagated up the call stack.

15.2.2.2. Propagating the **return** return code

Let us move on to a discussion of the return code value `2` / `return` which has its own subtleties. This code is returned either explicitly via the `return` statement or by the implicit `return` command at the end of every procedure body. It is the fundamental mechanism by which procedures return to their caller.

Practically all commands, except those specifically dealing with manipulation of return codes, such as `catch` (Section 15.4.1), `try` (Section 15.4.3) and the like, propagate the `return` code up the call stack like any other exceptional code. The special handling for the `2` / `return` code occurs in the case of evaluation of a procedure body, coroutine or `source` command.

In these cases, the procedure, coroutine or source evaluation will be terminated. However, unlike other commands, **rather than propagating this return code value of `return`, the procedure or script evaluation will complete with a return code `0`/`ok` so its caller sees a normal completion.**

Let us go through a concrete example of nested procedure calls to see how a return code of `2` / `return` is handled.

```
proc cmdB {} {
    return "a value"
    puts "cmdB returning"
}
proc cmdA {} {
    cmdB
    puts "cmdA returning"
}
cmdA
→ cmdA returning
```

In procedure `cmdB`, the `return` command itself completes with the result `a value` and a return code of `2` / `return`. Since this return code is something other than `ok`, execution does not continue with the subsequent `puts` statement. Rather the Tcl procedure evaluation implementation sees the return code `2` and treats it specially. The corresponding result `a value` is returned as the result of `cmdB` **and** instead of propagating the code `2` / `return`, a return code of `0` / `ok` is returned to the cmdA invocation. The `ok` return value allows the cmdA procedure to invoke its `puts` command before returning.

Note how the `2` / `return` return code is transformed to `0` / `ok` when the procedure completes. If the invocation of `cmdB` had propagated `2` / `return`, the procedure evaluation of `cmdA` would itself have returned immediately after `cmdB` returned without executing its `puts` command. As we will see later, it is also possible to accomplish the latter, effectively returning to the caller several levels up the stack.

Regarding the aforementioned subtleties with respect to the `return` return code, consider the following script.

```
proc cmdB {} {
    set x "cmdB"
    uplevel 1 {
        puts "x = $x"
        return
    }
    puts "cmdB returning"
}

proc cmdA {} {
    set x "cmdA"
    cmdB
    puts "cmdA returning" ❶
}
```

❶ Is this line printed?

What output would you expect when procedure `cmdA` is called? One might think that since the `uplevel` command called from `cmdB` executes in the context of `cmdA`, the `return` command within the `uplevel` would cause `cmdA` to return without printing the `cmdA returning` line. What actually happens is

```
% cmdA
→ x = cmdA
  cmdA returning
```

Based on our prior discussion, this behaviour should be, umm, obvious. If not, remember that all that the return statement does is to return the code `2` / `return`. Since `uplevel` itself does not treat this return code specially, it propagates to the **caller** of the `uplevel` command which is `cmdB`, **not** `cmdA`. On receiving this return code, the procedure invocation of `cmdB` terminates with a return code of `ok` as described above. The `cmdA` procedure thus only sees a `cmdB` return code of `ok` and continues on to invoke its `puts` command.

15.2.2.3. Propagating the **error** return code

The one standard exception code we have not discussed is `1` or `error`. This is because we have an separate section coming up soon devoted to error generation and handling in Tcl.

15.2.3. The return options dictionary

Along with the command result and return code, every command completion also includes a *return options dictionary*. This is a dictionary with keys

- `-code` and `-level` which together determine the return codes at each level of the call stack. These keys are not just informational but control the unwinding of the call stack and we describe how they are set and used in Section 15.3.
- `-errorinfo`, `-errorline`, `-errorcode` and `-errorstack`, which provide additional information when an error exception is raised. We detail these in Section 15.4.2.
- Any other application defined keys whose use and interpretation is entirely up to the application.

Note that only the keys `-code` and `-level` are guaranteed to exist on every completion.

We will be revisiting the return options dictionary as we proceed through the chapter. For now, we move on to a discussion of how they are generated alongside return codes.

15.3. The return command

```
return ?OPTION VALUE …? ? RESULT?
```

We have seen the use of the `return` command to return a result from a procedure. This is the most common use by far but the `return` command in Tcl is far more flexible and powerful than demonstrated by this typical use. It can be used to generate a return code, a custom return options dictionary and even skip levels in the call stack when returning from a procedure. We now explore its full functionality.

So far we have seen several commands that generate a specific return code:

- The `break`, `continue` commands generate their respective return codes.
- The `error` (Section 15.5.1) and `throw` (Section 15.5.1) commands that we describe later generate the `error` return code.
- Procedure invocations that complete normally do so with a return code of `ok`.

There is no way for these to generate other return codes and nor do they have any means of manipulating the return options dictionary.

The `return` command on the other hand provides a general-purpose mechanism to set all three values — the result, the return code and the return options dictionary — associated with a command completion. Additionally, the command has the ability to control the return codes generated at any level of the call stack, a facility that is important in construction of new control statements.

The `return` command **is just a command like any other** and as such it completes with the same three values — result, return code and return options dictionary — as any other command. Unlike other commands though, it allows the caller to specify the values to be returned for all three of these:

- The result of the `return` command is the *`RESULT`* argument which defaults to the empty string if unspecified.
- The return code from the `return` command itself is usually the code `2` / `return` but as we see in a bit, this can be controlled with the `-level` and `-code` options.
- The return options dictionary resulting from an call to `return` is composed from the specified *`OPTION VALUE`* pairs. As we stated in Section 15.2.3, this dictionary may have any number of keys. Tcl defines the names and semantics of certain keys but applications can add their own as well to pass along additional information. In addition, the key `-code` with a value of `2` / `return`, and the key `-level` with a value of `1`, are added to the dictionary if they are not already specified by the option value list. Finally, the option name `-options` is treated specially. Its value is treated as a dictionary whose content is merged with the other options to form the return options dictionary. We will see examples of custom return options dictionaries through the rest of the chapter.

Let us now take a closer look at the `-level` and `-code` options to the `return` command. We will start with a somewhat simplified explanation. The basic form of the `return` command

```
return "foo"
```

is equivalent to

```
return -code ok -level 1 "foo"
```

corresponding to the default values for the `-code` and `-level` options. With one important exception that we note below, the `return` command always completes with a return code of `2` / `return`. Note this is true irrespective of the value of the `-code` option. Since this return code is not a normal `ok` return code, when called within a procedure the procedure evaluation code will stop processing further commands in the procedure body. Moreover, since the procedure evaluation code affords special treatment to the code `return`, it will not be propagated as is. Rather the procedure evaluator treats the `return` code as a special case and completes the procedure itself with the return code value specified by the `-code` option, which is `ok` in our default case.

The sole exception we mentioned above with regards to the return code from the `return` command is when the `-level` option is specified with a value of `0`. In that case, the `return` command will itself complete with the code specified by the `-code` option and not with the `2` / `return` code.

To help clarify the difference in behaviour when `-level 0` option is specified versus the default `-level 1` value, let us contrast the following two procedures.

- `demo1` executes the equivalent of a normal return
- `demo0` executes a return with the `-level` option set to 0

```
proc demo1 {} {
    puts "demo1 enter"
    return -code ok -level 1
    puts "demo1 exit"
}
proc demo0 {} {
    puts "demo0 enter"
    return -code ok -level 0
    puts "demo0 exit"
}
```

Now if we invoke the two procedures, you see the difference in the output.

```
% demo1
→ demo1 enter
% demo0
→ demo0 enter
  demo0 exit
```

The behaviour of the `demo1` procedure is what you might expect. The second line is equivalent to the default `return` command and thus the `demo1 exit` line is not printed.

The `demo0` procedure behaviour is different. Despite the `return` command appearing before it, the second `puts` is still invoked and the `demo0 exit` line printed. This is explained by the fact that the `-level 0` option to `return` causes that command to complete with the return code specified by the `-code` option, which is `ok` in our example, instead of the `2` / `return` return code. The procedure evaluation sees this as a normal command completion, not an exception, and therefore continues with the next statement in the procedure, the `puts` command.

Under what circumstances might one use a value of `0` for the `-level` option? There are a couple that are commonly seen. One is when the `return` command is used, in lieu of the `error` or `throw` commands, to raise an error exception. This is discussed in Section 15.5.2.

The other common use of `-level 0` in Tcl is as an *identity* function whose result is simply the value passed in[1].

```
set n 0
set reciprocal [if {$n == 0} {return -level 0 Inf} else {expr {1/$n}}]
→ Inf
```

15.3.1. Unwinding multiple levels of the call stack

The `-level` option can be used to unwind multiple levels of procedure call stack, skipping intermediate callers. This requires us to detail exactly how a return code of `2` / `return` from a command is handled during evaluation of a procedure body.

When a command returns this return code **during a procedure evaluation**, the following sequence of steps are taken.

[1] The `string cat` command can also be used as an identity function

- First, the `-level` element of the return options dictionary is decremented.
- If the post-decrement value of `-level` is 0, the procedure completes and returns to its caller with completion return code being set to the value of the `-code` element in the return options dictionary. The caller of the procedure will handle this return code as appropriate. For example, if set to `ok`, it will continue with the next command.
- If the post-decrement value is greater than 0, the procedure is completed but with a return code of `2` / `return`, and **not** the code specified by the `-code` element of the return options dictionary. The caller of the procedure will see this return code and thus repeat this sequence of steps to handle it (assuming it is also a procedure). Note the propagated return options dictionary will contain the decremented value of `-level`.

The point in all this is that the `-level` command can be used to force a return **from any point in the call stack with any desired return code** as illustrated in the following example.

```
proc demo1 {levels} {
    puts "demo1 enter"
    demo2 $levels
    puts "demo1 exiting"
    return "demo1 return value"
}
proc demo2 {levels} {
    puts "demo2 enter"
    demo3 $levels
    puts "demo2 exiting"
    return "demo2 return value"
}
proc demo3 {levels} {
    return -level $levels "demo3 return value"
}
```

If we call `demo1` with an argument of `1`, the `return` command executed in `demo3` is essentially the default form of the command. As expected, all `puts` statements are executed and we can see the corresponding outputs as the call stack unwinds. The result of our `demo1` call is `demo1 return value`.

```
% demo1 1
→ demo1 enter
  demo2 enter
  demo2 exiting
  demo1 exiting
  demo1 return value
```

Now if we call `demo1` with an argument of `2`, we can see the output has changed.

```
% demo1 2
→ demo1 enter
  demo2 enter
  demo1 exiting
  demo1 return value
```

Notice that the `demo2 exiting` statement is no longer printed. This is a consequence of the command executed in `demo3` being

```
return -level 2 "demo3 return value"
```

Let us follow the earlier description of how the `-level` element is processed:

- The `return` command in `demo3` completes with a `-level` of `2`. This is decremented and since it is not `0`, the `demo3` procedure itself completes with a return code of `2` / `return` **and not** `0` / `ok` **as in the normal case**.
- The evaluation of `demo2` sees this `2` / `return` code and instead of executing the next command in the procedure as it would if the code were `ok`, it also completes as per the usual handling of the `return` return code. However, now the result of decrementing of the `-level` element is `0` and as per our `-level` processing rules, the procedure completes with a return code as specified by the `-code` element of the return options dictionary, which in this case is `0` / `ok`. The exiting `puts` statement as well as the final `return` command in `demo2` are never reached.
- The caller `demo1` sees this `ok` and moves on to processing the next command in its body. (Note that the **result** of `demo2` is not used and discarded.)

To go one step further, see what happens with the command

```
% demo1 3
→ demo1 enter
  demo2 enter
  demo3 return value
```

Now notice that neither the `demo1` nor the `demo2` exiting statements are printed. In addition, the result of the `demo1` command is the value originally returned from `demo3` and not the return value from `demo1` itself. You can extend the steps above to understand why that is.

Thus we see that the `return` command can be used to not only unwind the call stack but to also set the return value at that level. In case you were wondering, this also took place in our previous `demo1 2` example. The result of `demo3` was also the result of `demo2`. However, there the `demo1` procedure discarded the result of `demo2`.

In our illustration, we used the default `-code` value of `0` / `ok`. This is not mandated as shown in our next example where we write a utility command that checks if an argument is an integer and raises an error exception otherwise. We may write the procedure as follows:

```
proc check_integer {arg} {
    if {![string is integer -strict $arg]} {
        error "$arg is not an integer."
    }
}

proc tohex {arg} {
    check_integer $arg
    return [format %x $arg]
}
```

Passing it a non-integer raises an error exception.

```
% tohex abc
Ø abc is not an integer.
```

This works but the error stack (Section 15.4.2.1) is a bit messy.

```
% puts $::errorInfo
→ abc is not an integer.
      while executing
  "error "$arg is not an integer.""
      (procedure "check_integer" line 3)
      invoked from within
  "check_integer $arg"
      (procedure "tohex" line 2)
      invoked from within
  "tohex abc"
```

It is difficult to spot the line where the mistake was **made** as opposed to where it was **detected**. We can instead write the `check_integer` procedure as follows.

```
proc check_integer {arg} {
    if {![string is integer -strict $arg]} {
        return -level 2 -code error "$arg is not an integer."
    }
}
```

The error stack now looks much cleaner. We immediately know the call that needs to be fixed.

```
% tohex abc
Ø abc is not an integer.
% puts $::errorInfo
→ abc is not an integer.
      while executing
  "tohex abc"
```

15.3.2. Emulating other commands with return

Given our discussion of the `-code` option for the `return` command and an example of using it with the `error` return code, you might guess that the `return` command can be used to emulate other commands, like `break` or `continue`, that generate return codes. Here is a `stop` procedure that emulates the `break` command.

```
proc stop {} {return -code break}
```

And to show it works,

```
foreach char {a b c} {puts $char; stop} → a
```

The `stop` procedure effectively completes with a return code of `break`. The `foreach` command has special handling for this return code and duly terminates the loop. In a sense, commands like `break` and `continue` are really just syntactic sugar for specific common uses of the `return` command.

15.3.3. Custom return codes

Table 15.1 showed the return codes defined by Tcl ranging from 0 to 4. Applications and packages are free to use codes in the range 5 to 0x3FFFFFFF as a custom return code. Values outside this range are reserved for Tcl's future use. Custom return codes can be caught with `catch` or `try` just like any standard return code.

```
proc ret5 {result} {return -code 5 $result} → (empty)
catch {ret5 "Code 5!"} result               → 5
set result                                  → Code 5!
```

The interpretation of custom codes is of course entirely up to the application or package and different ones might interpret the same code differently. Of course, this can be a problem when multiple libraries are in use and thus such extensions must be used in very controlled fashion. See Section 15.7 for one possible use.

15.3.4. Custom return options dictionary

The other way return handling can be customized is through additional custom entries in the return options dictionary. This is done by passing the custom options to the `return` command. For example, to add a timestamp to errors before passing them on,

```
proc badcode {} { error "Did something bad!" }
proc demo {} {
    if {[catch {badcode} result ropts]} {
        return -options $ropts -timestamp [clock seconds] $result ❶
    } else {
        return $result
    }
}
```

❶ See Section 15.6.1 for an explanation of this idiom

The error is propagated with the timestamp added to the return options dictionary.

```
catch demo result ropts    → 1
set result                 → Did something bad!
dict get $ropts -timestamp → 1743694530
```

The above works because the `return` command will treat any option not known to it as a custom entry to be added to the return options dictionary.

15.4. Trapping exceptions

We saw in Section 15.2.2, how return codes are propagated up the call stack and how commands like `for` and `while` trap and handle special return codes like `break` and `continue`. We now describe the general purpose commands which can trap exceptions arising from **any** return code. Error exceptions are just a special case where the return code is `1` / `error`.

Let us start by looking at what happens when a command raises an error exception, i.e. it completes with a return code of `error`. As detailed in Section 15.2.2, all exception return codes are propagated up the call stack until handled by a command and the `error` return code is no exception (pardon the pun). If no such command appears in the call stack, the exception return code is propagated all the way up to the outermost level where the default error handler will terminate the program or, if it is a background error (Section 19.4.1) in code run from the event loop, an action like displaying an error message is invoked.

We saw how the looping constructs handle return codes of `break` and `continue` preventing the propagation of these codes up the call stack. Similarly, the `catch` and `try` commands allow the **trapping** of **any** return code from the execution of a command or script enabling further appropriate action to be taken.

15.4.1. Trapping exceptions: catch

```
catch SCRIPT ?RESULTVAR? ?OPTSVAR?
```

We will start off by looking at the `catch` command. The command executes the specified script returning as its result the **return code** from the script, **not the script result**.

```
catch {set x "Normal completion"}        → 0
catch {error "This is an error message"} → 1
catch {return "A result"}                → 2
catch {break}                            → 3
catch {continue}                         → 4
catch {return -code 5 -level 0}          → 5
```

If the *RESULTVAR* argument is specified, the variable of that name will hold the result of the script. The return code as before will be the result of the `catch` command itself. The result may come from a normal or exceptional completion.

```
catch {set x 100} result        → 0 ❶
puts $result                    → 100
catch {set x $nosuchvar} result → 1
puts "Error: $result"           → Error: can't read "nosuchvar": no such variable ❷
```

❶ Here `result` is the result of the script on normal completion
❷ Here `result` is the error message on an error exception

15.4.2. The error stack and return options dictionary

The optional *OPTSVAR* argument to the `catch` command is the name of a variable to hold the return options dictionary we discussed in Section 15.2.3. As we stated there, this dictionary always contains at least the two keys `-level` and `-code` that we have already elaborated on in previous sections. In the case of an error exception, the dictionary also contains the additional keys `-errorinfo`, `-errorcode`, `-errorstack` and `-errorline`.

Let us define a simple procedure that raises an error exception as a sample and print what we get back from the `catch`.

```
proc badproc {} { set y $nosuchvar } → (empty)
catch {badproc} result ropts         → 1
```

As we saw earlier, the `result` variable will hold the error message.

```
puts "result: $result" → result: can't read "nosuchvar": no such variable
```

The `ropts` variable holds additional information described in the following sections.

15.4.2.1. Error stack trace: -errorinfo element, errorInfo

The `-errorinfo` element contains a complete call stack dump that shows the sequence of calls up to the point the error exception was raised. The content is meant for human consumption and is primarily a debugging and troubleshooting aid.

```
% dict get $ropts -errorinfo
→ can't read "nosuchvar": no such variable
      while executing
  "set y $nosuchvar "
      (procedure "badproc" line 1)
      invoked from within
  "badproc"
```

This information is also stored in the `errorInfo` global variable.

```
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
      while executing
  "set y $nosuchvar "
...Additional lines omitted...
```

15.4.2.2. Error line number: -errorline element

The `-errorline` element gives the line number within the script body where the error was raised. Again, this is primarily for debugging purposes.

```
dict get $ropts -errorline → 1
```

15.4.2.3. Error codes: -errorcode element, errorCode

The `-errorcode` element in the dictionary contains additional information about the error in a form convenient for programmatic consumption. It is by convention structured as a list of elements, the first of which is the module generating the error followed by module specific information. In our example, it indicates the error was generated by Tcl itself, on a failed read operation on a variable. This information is also available via the `errorCode` global variable.

```
dict get $ropts -errorcode → TCL READ VARNAME
puts $::errorCode          → TCL READ VARNAME
```

The error code can often be parsed during execution to programmatically ascertain the specific cause and whether corrective action can be taken.

```
if {[catch {open options.ini} result ropts]} {
    if {[lindex $::errorCode 0] eq "POSIX" &&
        [lindex $::errorCode 1] eq "ENOENT"} {
        # File does not exist
        .. use default options ..
    } else {
        return -code error -options $ropts $result; # Propagate the error
    }
} else {
    # $result holds opened channel
    .. read options from $result ..
    close $result
}
```

15.4.2.4. Error stack: -errorstack element, info errorstack

The final error related element in the return dictionary is `-errorstack`.

```
dict get $ropts -errorstack → INNER loadScalar1 CALL badproc
```

This is similar to the `-errorinfo` element except it is in a form more suitable for programmatic consumption. It consists of alternating token and parameter pairs where the token may be one of `INNER`, `CALL` or `UP` indicating an internal command or byte code instruction, a procedure call, or a call frame change via `uplevel` (Section 14.1.5) and the like. The associated parameter gives the specifics. For example, in the above output, the `CALL` parameter indicates `badproc` as the name of the procedure that was called. It will also show the actual argument values in the invocation unlike `-errorinfo` which shows invocations before variable substitutions.

The content of the `-errorstack` element is also available with the `info errorstack` command. This returns the stack corresponding to the last error encountered.

```
info errorstack → INNER loadScalar1 CALL badproc
```

15.4.3. Trapping exceptions: try

```
try BODY ?HANDLER …? ?finally  FINALSCRIPT?
```

The `try` command offers a functionally equivalent alternative to the `catch` command (Section 15.4.1) for handling exceptions and errors. The latter is convenient when a single set of actions is to be taken on any non-normal return. The `try` command on the other hand makes it easier to break out handlers for different return codes or failure modes and when common set of actions, such as releasing resources, need to be taken for both normal and exceptional conditions. The choice is usually a matter of personal preference.

Each *`HANDLER`* specifies a completion status and/or an error code pattern along with a Tcl script. The command evaluates *`BODY`* and matches its completion status against the specification of each *`HANDLER`*. On finding a match, the corresponding handler script is evaluated. Remaining handlers are ignored.

If the `finally` clause is specified, the *`FINALSCRIPT`* argument is evaluated before the `try` command completes **irrespective of the completion status or whether any handlers were invoked**.

The completion status and result of evaluation of *`BODY`* is propagated as that of the `try` command unless a handler is executed in which case the completion status and result of the handler are propagated instead. If *`FINALSCRIPT`* completes normally, its result is thrown away. If it completes with an exception, that exception is propagated. In the cases where a handler or *`FINALSCRIPT`* generate an exception, the return options dictionary from the evaluation of *`BODY`* is added to the new return options dictionary under the `-during` key.

The *`HANDLER`* clauses themselves may take one of the following forms:

```
on CODE VARLIST HANDLERBODY
trap ERRORPATTERN VARLIST HANDLERBODY
```

For the `on` clause, *`CODE`* must be an integer or one of the mnemonic return code values shown in Table 15.1. The clause will match if the `try` command's *`BODY`* script completes with that return code.

A `trap` clause will match if *`BODY`* completes with a return code of `error` **and** *`ERRORPATTERN`* matches the return options dictionary's `-errorcode` element. *`ERRORPATTERN`* is treated as a list of words each of which must match the corresponding element of the `-errorcode` value. Additional elements in `-errorcode` are ignored so an empty *`ERRORPATTERN`* will match any value of `-errorcode`.

In both cases, *`VARLIST`* is a list of up to two variable names. On a match,

- the result of the evaluation of *`BODY`*, is assigned to the first name in *`VARLIST`*.
- the return options dictionary is assigned to the second name in *`VARLIST`*.
- the *`HANDLERBODY`* script is then executed. If *`HANDLERBODY`* is `-`, the *`HANDLERBODY`* of the next clause is used instead.

If a variable name is not present or is the empty string, the corresponding assignment is skipped.

The handlers are matched in sequence and the first match is evaluated. If no match is found, the return code and result are propagated (Section 15.2.2) to the caller.

A simple `try` without any additional clauses acts similar to `eval` (Section 3.13). If the *`BODY`* script completes normally, the result for the evaluation is the result of the `try` command. On any exceptional completions, the return code is propagated up **since the `try` command does not have any handlers specified**.

```
try {set x 1}          → 1
try {set x $nosuchvar} Ø can't read "nosuchvar": no such variable ❶
```

❶ Error return code is propagated up to the command shell

The `on` handlers can be used to trap completions with specific return codes. Any return codes that are trapped in this manner are **not** automatically propagated. We can use `catch` (Section 15.4.1) to examine the propagation of return codes.

```
% catch {
    try { error "Error!" } on error result {puts Trapped!}     ❶
}
→ Trapped!
  0
% catch {
    try { break } on error result {puts Trapped!}              ❷
}
→ 3
```

❶ Completes normally as `error` return code trapped.
❷ Propagates `break` as no handler defined for it.

Note that as for `catch` (Section 15.4.1), **any** return code can handled, even `ok`, `return` or non-standard numeric values.

The *`VARLIST`* argument lets us retrieve the result of evaluation of *`BODY`* and the return options dictionary in a similar manner to the two optional arguments to the `catch` command.

```
try {
    set x $nosuchvar
} on error {result ropts} {
    puts "result = $result"
    puts "Return options dictionary:"
    print_dict $ropts
}
→ result = can't read "nosuchvar": no such variable
  Return options dictionary:
  -code         = 1
  -errorcode    = TCL LOOKUP VARNAME nosuchvar
  -errorinfo    = can't read "nosuchvar": no such variable
...Additional lines omitted...
```

The other form that a `try` handler specification can take is specifically for trapping completions with an `error` return code. The utility of the `trap` handler over the `on error` handler is that it directly allows distinction between different error codes without having to separately check for them within an `on error` handler.

Consider the following commands.

```
expr {4/0}   Ø divide by zero
expr {4/0.0} → Inf
```

If we wanted the two to behave the same, we could define a `div` procedure.

```
% proc div {a b} {
    try {
        return [expr {$a/$b}]
    } trap {ARITH DIVZERO} result {
        return [expr {$a/0.0}]
    }
}
```

Our handler is invoked only when the return code is `error`, **and** the error code indicates an divided by zero. All other cases work as before including normal operation and other types of errors.

```
div 4 2   → 2
div 4 0.0 → Inf
div 4 0   → Inf ❶
div 4 xyz Ø cannot use non-numeric string "xyz" as right operand of "/"
```

❶ Returns same values as division by `0.0` instead of raising an error.

Note that an `on error` handler is equivalent to `trap {}` and handles all completions with a return code of `error`. Therefore any `trap` clauses should appear before an `on error` clause. Trap clauses placed after it will not have effect.

We are left with the `finally` clause to discuss. The most common use of this clause is to ensure that resources are freed irrespective of whether a script completes normally or not. Thus the clause is used in the fashion similar to the following.

```
set fd [open data.xml]
try {
    return [parse_data [read $fd]]
} finally {
    close $fd
}
```

The code ensures that the opened file channel is closed irrespective of any errors in reading or parsing the data.

15.5. Raising exceptions

In Section 15.3 we saw how we can specify any arbitrary value as the return code on completion of a script or procedure. Raising an exception is nothing other than specifying a value other than `0` / `ok` for the return code. Thus, we do not need to say anything more about this general case.

However, the `error` return code differs from other return codes in that Tcl takes some additional implicit actions such as generating the stack trace back we saw earlier. The following sections describe this special case.

15.5.1. Raising errors: throw, error

```
throw ERRORCODE MESSAGE
error MESSAGE ?ERRORINFO? ?ERRORCODE?
```

Tcl has two dedicated commands, `throw` and `error`, that always complete with a **return code** of `1` / `error`. The command **result** is the passed *`MESSAGE`* argument. *`ERRORCODE`* is the error code that will be stored in the `-errorcode` element of the return options dictionary and the `errorCode` global variable.

Because `throw` is a **relatively** new addition to Tcl, you will find `error` used more often. Moreover, it is tempting to be lazy and use the one argument form of `error`. However, it is now considered good practice to always specify an error code which makes `throw` syntactically a little more convenient and preferable for the common case where an initial value for the `errorInfo` stack trace is not required to be passed.

When an error exception is thrown, Tcl accumulates a stack trace of the calling sequence in the `-errorinfo` element of the return options dictionary and the `errorInfo` global. By default, this stack trace starts at the point of the call to `error` or `throw`. However, the `error` command allows specification of the *`ERRORINFO`* argument to "seed" the stack trace. We will see an example of its use for propagating a caught exception in a later section.

Raising an error exception is staightforward using either `throw` or `error`.

```
proc change_password {name pass} {
    set len [string length $pass]
    if {$len < 8} {
        throw [list OAUTH PASSLEN $len]  "Password length must be at least 8."
    }
    db_update $name $pass
}
```

A couple of points to note about the above example. The general convention for the format of the error code is a word that identifies the module or package (`OAUTH`), then one or more failure "reason" codes (`PASSLEN`) and possibly some detail about the error, in our case the length of the supplied password.

```
% change_password user abc
Ø Password length must be at least 8.
% puts $::errorCode
→ OAUTH PASSLEN 3
```

We could have replaced the `throw` command in the procedure with the equivalent `error` command.

```
error "Password length must be at least 8." "" [list OAUTH PASSLEN $len]
```

The last two arguments to `error` are optional so we could have also raised an error as

```
error "Password length must be at least 8."
```

In this case the error code is set to an empty string.

15.5.2. Raising errors: return -code

We described the `return` command in detail in Section 15.3. The command can also be used in lieu of `throw` and `error` to generate error exceptions. It offers more flexibility in that it permits additional elements to be added to the error dictionary as well as providing control of the error stack.

Here we detail additional considerations when using `return` for this purpose.

```
return -code error ?-errorcode ERRORCODE? ?-errorinfo ERRORINFO? \
    ?-errorstack ERRORSTACK? MESSAGE
```

The `-errorcode`, `-errorinfo` and `-errorstack` options set the corresponding element in the return options dictionary. *`ERRORCODE`*, *`ERRORINFO`* and *`MESSAGE`* have the same semantics as for the `throw` (Section 15.5.1) or `error` (Section 15.5.1) commands.

Here is a short example demonstrating the equivalent use of `throw` versus `return`.

```
proc check_boolean_1 {arg} {
    if {![string is boolean -strict $arg]} {
        throw {TYPECHECK BOOLEAN} "$arg is not a boolean"
    }
}
proc check_boolean_2 {arg} {
    if {![string is boolean -strict $arg]} {
        return -code error -errorcode {TYPECHECK BOOLEAN} \
            "$arg is not a boolean"
    }
}
```

If you run both procedures however, you will see a difference in the error stack.

```
% check_boolean_1 abc
Ø abc is not a boolean
% puts $errorInfo
→ abc is not a boolean
      while executing
  "throw {TYPECHECK BOOLEAN} "$arg is not a boolean""
      (procedure "check_boolean_1" line 3)
      invoked from within
...Additional lines omitted...
% check_boolean_2 abc
Ø abc is not a boolean
% puts $errorInfo
→ abc is not a boolean
      while executing
  "check_boolean_2 abc"
```

See Section 15.3.1 for another example of using `return` for a cleaner error stack.

15.6. Forwarding exceptions

There are circumstances where we need to trap an exception, handle it if we can, and if not, forward or re-throw the exception with the **same error code and error stack** as in the original. We can use the `return` or `error` command for this purpose.

15.6.1. Forwarding exceptions with return

The complete control you have over the result, return code, level and return options dictionary makes the `return` command ideal for forwarding any kind of exception.

Here is an example of its use for this purpose. The snippet calls a `recover` command to attempt to recover from an error. If the recovery fails, the **original** error and related state is propagated.

```
proc recover args {return 0}
proc do_something {} {
    set x $nosuchvar
}
proc demo {} {
    if {[catch {do_something} result ropts]} {
        if {![recover]} {
            return -options $ropts $result ❶
        }
    }
}
```

❶ Note there is no `-code` option specified because it is already contained in the `ropts` return dictionary

Let us confirm that the caught exception information is preserved.

```
% demo
Ø can't read "nosuchvar": no such variable
% puts $::errorCode
→ TCL READ VARNAME
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
       while executing
  "set x $nosuchvar"
       (procedure "do_something" line 2)
       invoked from within
...Additional lines omitted...
```

Notice that all information in the original exception is present. This includes cases where the returned option dictionary may contain custom elements.

15.6.2. Forwarding exceptions with error

Alternatively, we can use the `error` command in the special case where we want to forward error exceptions by specifying the original `errorCode` and `errorInfo` values as arguments to the `error` command.

```
proc demo {} {
    if {[catch {do_something} result]} {
        if {![recover]} {
            error $result $::errorInfo $::errorCode
        }
    }
}
```

If we invoke `demo`, notice again that the original error code and stack trace were preserved.

```
% demo
Ø can't read "nosuchvar": no such variable
% puts $::errorCode
→ TCL READ VARNAME
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
       while executing
  "set x $nosuchvar"
       (procedure "do_something" line 2)
       invoked from within
...Additional lines omitted...
```

This use of `error` to forward an error exception is seen in legacy code as **use of the `return` command for this purpose is preferred** for the following reasons:

- The `error` command cannot forward exceptions other than errors
- There is no means to preserve the full contents of the return options dictionary

15.7. Custom control statements

In Section 14.1.5 we implemented a new control statement `repeat`. That implementation was incomplete because it did not handle exceptional conditions like `break` and errors. Having described return code and error handling, we are now in a position to present a full implementation.

As a reminder, we want to implement a `repeat` command that can be used as follows

```
set sum 0
repeat i 10 {
    incr sum $i
}
```

The implementation is fundamentally the same as what we saw before except that we now handle exceptions from the evaluated script.

```
proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        set ret_code [catch {uplevel 1 $body} result ropts]
        switch $ret_code {
            0 {}
            3 { return }
            4 {}
            default {
                dict incr ropts -level
                return -options $ropts $result
            }
        }
    }
    return
}
```

The command now handles exceptions of all types.

- For `ret_code` values of `0` / `ok` and `4` / `continue`, we just continue with the next iteration of the loop.
- A value of `3` / `break`, we return from the procedure, thereby breaking out of the loop.
- For all other codes, which includes `2` / `return`, we propagate the result and return options dictionary back up the call stack.

There is a subtle point to be noted for this last case. The `-level` element of the return options dictionary is incremented before propagation. The reason for this is that the `repeat` command itself is a procedure so its return will result in the `-level` element being decremented (Section 15.3.1). The built-in iterators `for`, `while` etc. are not procedures and the `-level` element is not decremented on their completion. To emulate the built-in iterators, our `repeat` command must therefore increment `-level` to compensate for the additional decrement.

Having implemented a custom loop command let us extend its functionality further with another control command, `skip` that behaves like `continue` but lets you specify how many iterations of the loop are to be skipped.

```
proc skip {skip_count} { return -code 5 $skip_count }
```

We have now introduced a new return code `5` and have to account for it in our `repeat` procedure.

```
    proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        set ret_code [catch {uplevel 1 $body} result ropts]
        switch $ret_code {
            0 {}
            3 { return }
            4 {}
            5 { incr iter $result }
            default {
                dict incr ropts -level
                return -options $ropts $result
            }
        }
    }
    return
}
```

We can now use it to skip a given number of iterations.

```
repeat n 5 {
    puts "Iteration $n"
    if {$n == 1} {skip 2}
}
→ Iteration 0
  Iteration 1
  Iteration 4
```

Of course, only our `repeat` custom iteration command understands this new `skip` construct. It cannot be used with built-in iterators which will treat the unrecognized return code `5` as an error.

16

Namespaces

> I, sir, am Dromio; command him away.
>
> I, sir, am Dromio; pray, let me stay.
>
> — William Shakespeare *Comedy of Errors*

If Shakespeare understood namespaces, there would have been no confusion between Syracuse::Dromio and Ephesus::Dromio. Much havoc could have been avoided!

Most modern languages support the concept of *namespaces* as a means to resolve conflicts between multiple libraries or components defining the same name for a variable, function or any programming construct. This is even more of an issue for dynamic and scripting languages where there is no separate compile/link step that can be used to limit name visibility to file scope. A common convention before the advent of namespaces was to prefix names with the name of the module or library so `libA_state` and `libB_state` could be distinguished. Given that most references to names are to those within the same module, this is not just unnecessary typing but a hindrance to readability as well.

Namespaces are a solution to this issue. They provide a means to partition names and define a scope within which they are visible so there is no confusion as to which variable, command or other programming construct is being referenced.

Tcl's support for namespaces is dynamic and scriptable. It goes further than most languages in its capabilities and flexibility. We explore these features in this chapter.

16.1. Namespace basics

A *namespace* is a mechanism for grouping together variables and commands under an identifier, the name of the namespace. It also creates a *scope* for execution of code wherein names within the same namespace can be referenced without further qualification while requiring names outside the namespace to be qualified with the name of their containing namespace.

16.1.1. A simple namespace example

A simple script will clarify the concepts.

```
namespace eval nsA {
    variable my_var "variable in nsA"
}
```

The above command creates a namespace `nsA` and evaluates the passed script within the *context* of the namespace. Thus the script

```
variable my_var "variable in nsA"
```

is executed in the context of namespace `nsA`.

The command `variable` is used to declare and optionally initialize a variable inside the namespace context within which it is executed.

Next we will create another namespace, `nsB`, in the same fashion.

```
namespace eval nsB {
    variable my_var "variable in nsB"
}
```

And finally we will call `variable` **outside** of any namespace.

```
variable my_var "global variable"
```

This `variable` command is executed outside of any namespace and hence defaults to the global namespace context. The variable is a global variable similar to that created by the `global` command (Section 3.6.5.2).

We are now ready to give some examples of scope and context.

```
puts $my_var → global variable
```

Because the `puts` is executing outside any namespace, or to be precise, in the global namespace, the reference `my_var` is to the global variable of that name. On the other hand, if the same command were to be executed within the context of namespace `nsB`

```
namespace eval nsB { puts $my_var } → variable in nsB
```

the name `my_var` would refer to the variable defined in the `nsB` context.

In both cases, the variable references were *unqualified* and hence defaulted to the namespace context in which the code was executing. To refer to variables **outside** the context in which the code is executing, the name must be *qualified* with the name of the containing namespace. For example, to access the variables in the global and `nsA` namespace contexts from code running under the `nsB` context,

```
% namespace eval nsB {
    puts $::my_var
    puts $::nsA::my_var
}
→ global variable
  variable in nsA
```

Although the above example used variables to demonstrate context, the same also applies to command definitions and invocations as we will see as we proceed.

It is completely legal and often very useful to store a namespace name itself in a variable and use the variable to refer to the contents of the namespace. However, you have to be careful in the syntax used.

```
% set my_namespace nsA
→ nsA
% puts $my_namespace::my_var
Ø can't read "my_namespace::my_var": no such variable
```

The above generates an error because the parser treats `my_namespace::my_var` as the name of the variable and tries to resolve it. You need to therefore use one of several alternative syntaxes instead.

You can use the `${}` syntax to constrain the parsing of the variable name and then use `set` to retrieve the value.

```
puts [set ${my_namespace}::my_var] → variable in nsA
```

Alternatively, you can use nested `set` commands.

```
puts [set [set my_namespace]::my_var] → variable in nsA
```

Finally, to save repeated typing, you can link a local variable to the namespace variable using the `namespace upvar` command described in Section 16.5.1.3.

```
namespace upvar $my_namespace my_var linked_var → (empty)
puts $linked_var                                → variable in nsA
```

16.1.2. Namespace names and hierarchy

Namespaces may nest in a hierarchical fashion similar to the paths in a file system except for the use of `::` as the separator instead of `/` or `\`. So for example, the identifier `a::b::c` consists of the namespace `a`, the namespace `b` contained within `a`, and an identifier `c` which may be a variable, command name or even another namespace inside `a::b`. The root of the hierarchy is the global namespace whose name is the empty string so `::a` refers to an identifier `a` in the global namespace.

Just like file paths, names may be absolute or relative. An absolute name always starts with a `::` and defines a path through the namespace hierarchy starting with the global namespace. An example is `::a::b::c`. A relative name does not start with a `::` and defines a path through the namespace hierarchy relative to the namespace in which it is referenced. The name `a::b::c` is a relative name and is **not** the same as `::a::b::c` unless the reference occurs in the global namespace.

Let us rework our earlier example to include nested namespaces. The `namespace current` command, which we will see later, returns the name of the current namespace context.

```
namespace eval nsA {
    variable my_var "[namespace current] variable"
    namespace eval nsB {
        variable my_var "[namespace current] variable"
    }
}
namespace eval nsB {
    variable my_var "[namespace current] variable"
}
variable my_var "[namespace current] variable"
```

With these definitions in place, we can see how the different variables might be referenced from within the `nsA` namespace.

```
namespace eval nsA {puts $my_var}         → ::nsA variable ❶
namespace eval nsA {puts $::my_var}       → :: variable ❷
namespace eval nsA {puts $nsB::my_var}    → ::nsA::nsB variable ❸
namespace eval nsA {puts $::nsB::my_var}  → ::nsB variable ❹
```

❶ Current namespace
❷ `::` is a synonym for the global namespace
❸ Relative namespace
❹ Absolute namespace

We will have more to say about name resolution in Section 16.5.

There are a couple of points to be noted about nested namespaces.

First, a nested namespace can be directly defined with a single `namespace eval` so that instead of

```
namespace eval nsA {
    namespace eval nsB {
        .. Some code ..
    }
}
```

we could have said

```
namespace eval nsA::nsB {
    .. Some code ..
}
```

which would have resulted in the whole hierarchy being created if necessary.

The other point to be noted is that each `namespace eval` for a namespace does not overwrite any existing namespace of that name; it modifies or adds to it as we saw in the above examples.

16.1.2.1. Inspecting namespace hierarchies: namespace current|parent|children

```
namespace current
namespace children ?NAMESPACE?
namespace parent ?NAMESPACE?
```

The `namespace current` command returns the current namespace context.

```
namespace eval nsA {
    proc whereami {} {return [namespace current]}
}
puts [nsA::whereami]
→ ::nsA
```

In the above fragment, the `proc` definition is inside the `nsA` namespace and consequently, the `whereami` procedure is also created within that namespace.

The `namespace parent` command returns the fully qualified name of the namespace containing a specified namespace. If *`NAMESPACE`* is not specified, it defaults to the namespace from which the command is invoked.

```
namespace eval nsA { namespace parent }     → :: ❶
namespace eval nsA { namespace parent nsB } → ::nsA ❷
namespace parent nsB                        → :: ❸
namespace parent ::                         → (empty)
```

❶ Parent of current namespace context
❷ `nsB` child of `nsA`
❸ `nsB` child of global namespace

Conversely, the `namespace children` command returns a list of namespaces that are the children of a specified namespace. Again, *`NAMESPACE`* defaults to the current namespace if unspecified.

```
namespace eval nsA {namespace children} → ::nsA::nsB
namespace children nsA                  → ::nsA::nsB
namespace children ::
 → ::twapi ::platform ::zlib ::nsA ::pkg ::oo ::nsB ::tcl
```

16.1.2.2. Manipulating names: namespace qualifiers|tail

```
namespace qualifiers IDENTIFIER
namespace tail IDENTIFIER
```

Unlike static languages, programs in dynamic languages like Tcl often construct namespaces on the fly at runtime. Tcl therefore provides some commands to make manipulation of namespace names easier.

The `namespace qualifiers` command returns the leading namespace qualifiers from an identifier. Correspondingly, `namespace tail` returns the last component of an identifier. Both commands work purely on a syntactic basis. There is no requirement for the namespaces in *IDENTIFIER* to actually exist.

```
set nshead [namespace qualifiers ::no::such::namesp] → ::no::such
set nstail [namespace tail ::no::such::namesp]       → namesp
```

The above commands deconstruct an identifier into namespace components. There are no complementary commands to *construct* namespace paths because use of normal string interpolation or commands like `join` (Section 4.13) is sufficient.

```
set my_ns "::${nshead}::$nstail"            → ::::no::such::namesp
set my_ns [join [list $nshead $nstail] ::]  → ::no::such::namesp ❶
```

❶ Useful when the namespace components are already in list form

When constructing namespace identifiers, it is useful to know that Tcl will treat more than two `:` characters as namespace separators as well.

```
puts $::nsA:::::::::my_var → ::nsA variable
```

Thus when interpolating or joining you do not need to worry about trailing namespace separators in the identifier.

16.1.3. Deleting a namespace: namespace delete

```
namespace delete ?NAMESPACE …?
```

As is always the case with Tcl, program elements can be created and destroyed at will and namespaces are no exception. We have seen how namespaces are created with `namespace eval`. The complementary command to destroy namespaces is `namespace delete`.

The command takes zero or more namespace names and deletes each along with all its contained program elements, including variables, commands and even nested namespaces.

16.1.4. Checking namespace existence: namespace exists

```
namespace exists NAMESPACE
```

Given that they can appear and disappear on the fly, there needs to be a means of checking whether a namespace exists. The `namespace exists` command returns `1` if the specified namespace exists and `0` otherwise.

16.2. Executing code in a namespace: namespace eval|inscope

```
namespace eval NAMESPACE SCRIPT ?SCRIPT …?
namespace inscope NAMESPACE SCRIPT ?ARG …?
```

We have already seen how code is executed in the context of a namespace with the `namespace eval` command.

The command will create the namespace of the specified name if it does not already exist. It then concatenates the remaining arguments, separating them with spaces, and evaluates the result in that namespace.

`NAMESPACE` is resolved as we detail later in Section 16.5.2. If it is a hierarchical namespace, intermediate namespaces are created as necessary. So for example,

```
namespace eval ns1 {
    namespace eval ns2::ns3 {}
    namespace eval ::ns4 {}
}
```

evaluated in the global scope will create namespaces `::ns1`, `::ns1::ns2`, `::ns1::ns2::ns3` and `::ns4`.

What does execution in the context of a namespace mean? It has primarily to do with how names (variables, commands, namespace names) are resolved as we summarized earlier and will go into detail in Section 16.5.

The `namespace inscope` command is very similar to `namespace eval`. Like `namespace eval`, `namespace inscope` will execute *`SCRIPT`* in the context of the specified namespace but with two important differences:

- The first is that `namespace inscope` will **not** create the namespace if it does not already exist.
- The other difference is that the arguments are not all concatenated before execution as is done by the `namespace eval`. Rather, *`SCRIPT`* is executed after appending the remaining arguments as proper list elements. In effect, the additional arguments do not undergo a second round of substitution as is the case with `namespace eval`.

The following code snippet illustrates the difference. First, a small procedure to print arguments defined inside a namespace:

```
namespace eval ns1 { proc print_args {args} {puts [join $args ,]} }
```

Now we evaluate a call to the procedure via both `namespace eval` and `namespace inscope`. As seen from the output, the arguments undergo two rounds of substitution with `namespace eval` and only one with `namespace inscope`.

```
% set arg1 "First argument"
→ First argument
% set arg2 {$::arg1}
→ $::arg1
% namespace eval ns1 { print_args } $arg1 $arg2
→ First,argument,First argument
% namespace inscope ns1 { print_args } $arg1 $arg2
→ First argument,$::arg1
```

The `namespace inscope` command is rarely used directly in Tcl programming. Rather its primary purpose is to form the basis of the `namespace code` command which serves a specific purpose that we describe next.

16.2.1. Namespace contexts in callbacks: namespace code

```
namespace code SCRIPT
```

Tcl programming often involves callbacks — scripts and commands that are invoked from the event loop or other contexts. For such cases, callback scripts that expect to be run in the context of a specific namespace will fail, for example the following snippet:

```
namespace eval ns1 {
    variable avar "Some value"
    after 100 {puts $avar}
}
```

This will fail because the callback script `puts $avar` will execute in the global context where there is no variable `avar` defined. We really want to execute the script in the context of `ns1`. The `namespace code` provides a convenient mechanism to accomplish this.

The command result is a script that can be evaluated in **any** scope, global or any other namespace, and will still result in *`SCRIPT`* being invoked in the same namespace context in which the `namespace code` was invoked.

So the above fragment would work correctly if written as

```
namespace eval ns1 {
    variable avar "Some value"
    after 100 [namespace code {puts $avar}]
}
```

You can examine the result of the command to see how it works. The passed script is wrapped in `namespace inscope` to achieve the desired result.

```
% namespace eval ns1 { namespace code {puts $avar} }
→ ::namespace inscope ::ns1 {puts $avar}
```

The following utility procedure is useful as syntactic sugar for capturing the namespace scope when the callback script consists of a single command.

```
proc callback {args} {tailcall namespace code $args}
```

Then the above call can be written as

```
after 100 [callback puts $avar]
```

16.3. Namespace variables: variable

```
variable ?NAME VALUE …? ? NAME?
```

There are two ways a variable can be defined within a namespace. The first, and **recommended**, way is through the explicit use of the `variable` command. The command takes a list of alternating variable name and value arguments with the value for the last variable name being optional.

The command may be invoked directly from a `namespace eval` script or from within a procedure. In the former case,

- if a variable of a specified name does not exist, it is created. If a corresponding initializing value is specified, it is assigned to the variable. Otherwise, the variable is created but left undefined (Section 3.6.5.3).
- if the variable of that name already existed within the namespace, it is assigned the initializing value if specified and left unaltered otherwise.

When the command is invoked from a procedure, the behaviour is similar except that the command creates variables local to the procedure but linked to namespace variables of the same name. This is detailed in Section 16.5.1.2 when we discuss name resolution in procedures.

An example of `variable` use.

```
namespace eval nsA {
    variable var_a "abc"
    variable var_b [clock seconds] var_c
    variable var_d
}
```

Here the variable `var_a` and `var_b` are created (assuming they did not already exist) and initialized while `var_c` and `var_d` are created but remain undefined.

```
set nsA::var_a         → abc
info exists nsA::var_c → 0
```

The other way to create namespace variables, mentioned here for completeness but **not recommended**, is to directly assign to them from within the namespace context without explicitly declaring them with the `variable` command.

```
namespace eval nsA {
    set var_e 42
}
puts $::nsA::var_e
→ 42
```

It is strongly recommended that namespace variables be explicitly created with `variable`, particularly in code that supports both Tcl 8 and Tcl 9. In Tcl 8, without a preceding `variable` statement, setting a variable would overwrite a global variable of the same name if one existed. This acknowledged misfeature in Tcl 8 was corrected in Tcl 9.

16.4. Defining commands in a namespace

So far our examples have dealt with defining variables in namespaces. We now look at the same for defining procedures.

If the name passed to `proc` is fully qualified, it defines the containing namespace no matter where the procedure definition is placed.

```
namespace eval nsA::nsB {} ❶
proc ::nsA::nsB::demo_a {} {return [namespace current]}
namespace eval nsC {
    proc ::nsA::nsB::demo_b {} {return [namespace current]}
}
puts "[::nsA::nsB::demo_a], [::nsA::nsB::demo_b]"
→ ::nsA::nsB, ::nsA::nsB
```

❶ Make sure the namespaces exist

If the name does not have any namespace qualifiers or is not fully qualified, it is treated as relative to the current namespace.

```
namespace eval ::nsA {
    proc demo_c {} {return [namespace current]} ❶
    proc nsB::demo_d {} {return [namespace current]} ❷
    puts [demo_c]
    puts [nsB::demo_d]
}
→ ::nsA
  ::nsA::nsB
```

❶ Defined in namespace `::nsA`
❷ Defined in namespace `::nsA::nsB`

Other Tcl commands, such as TclOO object constructors, which create new commands also behave as above. The notable exception is `interp alias` (Section 23.6) which always resolves

relative names in the context of the global namespace even when invoked from within another namespace.

16.4.1. Namespace contexts in procedures

When a procedure is executed, its code runs in the context of the namespace in which the procedure is defined. Use of `variable` inside the procedure ties the specified name to the variable of the same name in the procedure's namespace. Calls to other procedures that are not fully qualified first look up procedures defined in the same namespace context.

```
proc demo {} {return "Proc in [namespace current]"}
namespace eval nsA {
    variable my_var "Variable in [namespace current]"
    proc demo {} {return "Proc in [namespace current]"}
    proc test_proc {} {
        variable my_var
        puts "Calling namespace proc: [demo]"
        puts "Calling global proc: [::demo]"
        puts "Value of my_var=$my_var"
    }
}
nsA::test_proc
→ Calling namespace proc: Proc in ::nsA
  Calling global proc: Proc in ::
  Value of my_var=Variable in ::nsA
```

16.5. Name resolution

When a name being referenced in a script begins with a `::` sequence, it is a fully qualified, or absolute name that uniquely identifies its target by specifying a path through the namespace hierarchy starting at the root (global) namespace. These names do not need to be resolved and further discussion in this section only pertains to names that are not fully qualified.

For relative names, both simple names that have no `::` separators as well as those that do but do not begin with `::`, the manner of resolution depends on whether the name corresponds to a variable, a namespace or a command.

16.5.1. Resolving variable names

We will first look at how names of variables are resolved in various contexts.

16.5.1.1. Variable resolution outside a procedure

Both simple name references and names that are not fully qualified are first resolved in the current namespace and an error raised if not found.

Tcl 8.6 would also attempt to resolve names in the global namespace if not found in the current namespace. This is no longer the case in Tcl 9.

16.5.1.2. Variable resolution in a procedure

Variable name resolution within a procedure is slightly different because there are procedure-local and argument names to deal with. Moreover, there are differences between resolution of simple names, i.e. names without any `::` separators, and names which have at least one namespace component (but are not fully qualified).

A simple name is local to the procedure, or an argument, unless previously linked via a `variable` (Section 16.3) or `upvar` (Section 14.1.4) command. If linked via `variable`, it is linked to the variable of the same name defined in the context of the namespace in which the procedure is defined. In the case of `upvar` it is linked to a variable defined further up the call stack (Section 14.1.4).

A variable that is a relative name with namespace components is resolved within the namespace in which the procedure resides.

The following example illustrates the different cases. Assume we have the following namespace structure.

```
set my_var "global variable"
namespace eval nsC {
    variable my_var "nsC variable"
}
namespace eval nsA {
    variable my_var "nsA variable"
    namespace eval nsB {
        variable my_var "::nsA::nsB variable"
    }
}
```

The various ways names are resolved is illustrated in the following procedure.

```
proc nsA::demo {} {
    variable my_var ❶
    set local_var "local"
    puts "local_var = $local_var"          ❷
    puts "my_var = $my_var"            ❸
    puts "nsB::my_var = $nsB::my_var"  ❹
}
nsA::demo
→ local_var = local
  my_var = nsA variable
  nsB::my_var = ::nsA::nsB variable
```

❶ Creates a local `my_var` linked to `::nsA::my_var`
❷ Variable local to procedure
❸ Variable linked to `::nsA::my_var`
❹ Relative name successfully resolved from current namespace

16.5.1.3. Linking to namespace variables: namespace upvar

```
namespace upvar NAMESPACE ?NSVAR LOCALVAR …?
```

The `namespace upvar` command allows linking of variables in a procedure-local or namespace context to variables in any target namespace.

The *`NAMESPACE`* argument specifies the name of the target namespace. It is resolved as described in Section 16.5.2.

Each *`NSVAR LOCALVAR`* pair specifies a variable name in the *`NAMESPACE`* namespace and the corresponding variable in the current context that will be linked to it. When the command is invoked outside a procedure, *`LOCALVAR`* names a namespace variable in the namespace in which the command is invoked. Below, `::nsC::linked_var` is linked to `::nsA::nsB::my_var`.

```
namespace eval nsC {
    namespace upvar ::nsA::nsB my_var linked_var
}
puts $::nsC::linked_var
→ ::nsA::nsB variable
```

Within a procedure, *`LOCALVAR`* is a procedure-local variable.

```
proc demo {} {
    namespace upvar ::nsA my_var linked_var
    puts $linked_var
}
demo
→ nsA variable
```

In all cases, the *`LOCALVAR`* variable must not already exist.

16.5.2. Resolving namespace names

Resolution of namespace names that are not absolute is very simple. They are always resolved with respect to the current namespace.

```
namespace eval nsA {
    namespace eval childNS {}
}
```

The unqualified name `childNS` results in creation of a namespace of that name in the current namespace context, ie. `nsA::childNS`.

16.5.3. Resolving command names

Resolution of command names differs from that of variable and namespace names in that additional mechanisms, *name imports* and *namespace paths*, are available to control the ways names are resolved.

Resolution proceeds in the following manner:

1. The current namespace is checked first.
2. If not found there, all namespaces on the namespace path, which is a list of namespaces, are checked in the order of their appearance.
3. If the command is still not found, the global namespace is looked up.
4. As a final resort, the `namespace unknown` handler is called.

In steps 1-3 above, a command may exist in a namespace either because it is defined there **or because it has been imported into that namespace**.

16.5.3.1. Importing names: namespace export|import|forget

```
namespace export ?-clear? ?PATTERN …?
namespace forget ?PATTERN …?
namespace import ?-force? ?PATTERN …?
```

Namespace export and import is a convenience feature that allows a namespace to mark selected commands as exported and callable without requiring any namespace qualifiers from any other namespace that imports them.

The namespace from where commands are being exported uses one or more `namespace export` commands to designate the commands to be exported. Without arguments, the command returns the list of names that are exported from the current namespace. Otherwise the list of *`PATTERN`* arguments are **appended** to the current list of exported patterns. Any command whose name matches any pattern in this export list using `string match` rules (Section 4.24) can be imported into other namespaces. If the `-clear` option is specified, the export list is emptied before the patterns are added.

The complement to `namespace export` is `namespace import` which is invoked from the namespace into which commands are to be imported. If no arguments are present, the command returns a list of the commands that have been imported into the current namespace. Otherwise, for every command that matches any of the *`PATTERN`* arguments a new command is created in the current namespace that points to the original command. *`PATTERN`* may be fully or partially qualified but only the last component is treated as a pattern. Qualifiers, if any, are treated as literals.

By default, if there is an existing command of the same name as the command being imported, an error is generated. If the `-force` option is specified, then instead of generating an error, the imported command overwrites the existing one.

Here is an example illustrating the basic working of export and import of names.

```
namespace eval nsA {
    proc aproc {} {puts "aproc called"}
    proc bproc {} {puts "bproc called"}
    proc cproc {} {puts "cproc called"}
    namespace export a* b*
}
namespace eval nsB { namespace import {::nsA::[ac]*} }
```

Let us check what commands actually land up being exported and imported.

```
namespace eval nsA { namespace export } → a* b*
namespace eval nsB { namespace import } → aproc
```

When we invoke commands in `nsA` from `nsB` using unqualified names,

```
namespace eval nsB { aproc } → aproc called
namespace eval nsB { cproc } Ø invalid command name "cproc" ❶
namespace eval nsB { bproc } Ø invalid command name "bproc" ❷
```

❶ Fails because `cproc` is not exported from nsA
❷ Fails because `bproc` is not imported into nsB

The `namespace export` command is "sticky" in that even a new command defined **after** it has been executed will also be exported if its name matches a pattern in the list of exported command patterns. On the other hand, the `namespace import` command only imports those commands that already existed at the time it was invoked. For example, let us define a new command that matches a pattern we previously exported.

```
namespace eval nsA {
    proc acommand {} { puts "acommand called" }
}
```

Now let us invoke it from `nsB`.

```
% namespace eval nsB { acommand } ❶
Ø invalid command name "acommand"
% namespace eval nsB { namespace import {::nsA::[ac]*} }
% namespace eval nsB { acommand } ❷
→ acommand called
```

❶ Fails because `namespace import` takes a snapshot
❷ Succeeds after we invoke `namespace import` again

We did not have to re-export the command but we did have to do a re-import.

Imported commands can be re-exported from the importing namespace.

```
namespace eval nsB { namespace export aproc }
namespace eval nsC {
    namespace import ::nsB::aproc
    aproc
}
→ aproc called
```

You can undo the effect of a `namespace import` with the `namespace forget` command. The *`PATTERN`* arguments are of the same form as accepted by `namespace import` except that they can also be simple names without namespace qualifiers. If namespace qualifiers are present,

the argument is matched against exported commands from all matching namespaces and the commands imported into the current namespace, if any, are removed. If no namespace qualifiers are present, any command matching the pattern in the current namespace are removed if they were imported.

Either of the following would undo the effect of the `namespace import` into `nsB`.

```
namespace eval nsB { namespace forget ::nsA::aproc }
namespace eval nsB { namespace forget acommand }
```

As we can see both commands are removed from `nsB`.

```
namespace eval nsB { aproc }    Ø invalid command name "aproc"
namespace eval nsB { acommand } Ø invalid command name "acommand"
```

16.5.3.2. Namespace paths: namespace path

```
namespace path ?NAMESPACELIST?
```

An alternate way to set up access to another namespace's commands without requiring qualification for every call is through namespace paths. A *namespace path* is a list of namespaces that should be searched to locate a command if it is not found in the current namespace. This list is specific to a namespace and can be set up with `namespace path`.

If *`NAMESPACELIST`* is specified, it should be a list of namespace names and the namespace path for the context in which the command is called is set to this value. If no argument is specified, the command just returns the namespace path for the current namespace.

The example below illustrates several points about namespace paths.

```
proc global_proc {} {puts "global_proc called"}
namespace eval nsA {
    proc nsA_proc {} { puts "nsA_proc called" }
    namespace eval nsB { proc nsB_proc {} { puts "nsB_proc called" } }
}

namespace eval nsC {
    namespace path [list ::nsA ::]
    puts "The namespace path is now [namespace path]."
    proc nsC_proc {} { nsB::nsB_proc }
    global_proc
    nsA_proc
    nsC_proc
}
→ The namespace path is now ::nsA ::.
  global_proc called
  nsA_proc called
  nsB_proc called
```

Note from this example that

- The global namespace is like any other namespace and can be explicitly placed at any position on the namespace path for a namespace if so desired. Keep in mind though that commands in the global namespace will automatically be resolved in any context even if they do not appear on the namespace path. Adding it to the path only makes sense if you want it to be searched **before** some of the namespaces in the path.
- The namespace path is searched not only for simple names but for relative names with namespace components.
- The namespace path is effective not only within a `namespace eval` but also within procedures defined in that namespace.

16.5.3.3. Comparing namespace imports and paths

Though similar in their ability to reference program elements in one namespace from another without explicit qualification, the namespace import/export and path features work differently.

Importing a name into a namespace with `namespace import` actually **creates** a command in that namespace which points to the command in the exporting namespace. On the other hand, `namespace path` does **not** create a new command. The following example will clarify the differences.

```
namespace eval nsA {
    proc aproc {} { puts "aproc called" }
    namespace export aproc
}
namespace eval importer { namespace import ::nsA::aproc }
namespace eval pathfinder { namespace path ::nsA }
```

The command `nsA::aproc` can be accessed from both namespace without qualification.

```
namespace eval importer { aproc }   → aproc called
namespace eval pathfinder { aproc } → aproc called
```

However, the two are not equivalent as the following fragments illustrate.

```
importer::aproc   → aproc called
pathfinder::aproc Ø invalid command name "pathfinder::aproc"
```

In the first case, `importer::aproc` can be directly called because importing actually creates a command of that name in `importer`. The second call raises an error because there is no `aproc` in `pathfinder` and the namespace path only applies to commands invoked from within `pathfinder`. To confirm,

```
info commands importer::*   → ::importer::aproc
info commands pathfinder::* → (empty)
```

Here is a slightly different effect of the same.

```
namespace eval nsB { namespace path ::importer }
namespace eval nsC { namespace path ::pathfinder }
```

If we try to invoke `aproc` from the `nsB` and `nsC`, the first works and the second does not.

```
namespace eval nsB { aproc } → aproc called
namespace eval nsC { aproc } Ø invalid command name "aproc"
```

Another way of viewing the difference is that imports link to the original command whereas the path mechanism searches for the command by name along the search path. Thus if we were to rename the original command or the one in the importing namespace, imports would continue to work.

```
rename ::nsA::aproc ::nsA::aproc2                  → (empty) ❶
namespace eval importer {aproc}                    → aproc called
rename ::importer::aproc ::importer::a_better_name → (empty) ❷
importer::a_better_name                            → aproc called
```

❶ Rename the original procedure
❷ Rename the procedure in the importing namespace

On the other hand, the path mechanism would no longer locate a command of that name in the defining namespace.

```
% namespace eval pathfinder {aproc}
Ø invalid command name "aproc"
```

16.5.3.4. Handling unknown commands: namespace unknown

```
namespace unknown ?COMMANDPREFIX?
```

If all the mechanisms discussed above fail to resolve a command, Tcl will call the *unknown command handler* for the namespace. This handler is set independently for each namespace by calling the `namespace unknown` command from the context of the namespace to which it is to be applied.

If specified, *`COMMANDPREFIX`* should be a list comprising the name of a command and zero or more arguments. When a command cannot be resolved within the namespace, the entire command including arguments is appended to *`COMMANDPREFIX`* and invoked. The result is then returned as the result of the original command.

In our example, if a command is not located when called from the `nsA` namespace, we will try invoking it as an external program instead.

This example is for pedagogic purposes only. It is not safe programming practice!

```
% namespace eval nsA { ls *.adocgen }
Ø invalid command name "ls"
% namespace eval nsA { namespace unknown [list exec -keepnewline --]}
→ exec -keepnewline --
% namespace eval nsA { ls *.adocgen}
→ basics.adocgen
  binary.adocgen
...Additional lines omitted...
```

If no arguments are passed, the command returns the current handler.

```
% namespace eval nsA {namespace unknown}
→ exec -keepnewline --
```

Note that the handler for the namespace is only executed when a command lookup fails within the specified namespace context. It will not be invoked either when lookups fail in some other context or even when an attempt is made to call a non-existent command within the handler's context from outside the context. Thus neither of the following will invoke our handler.

```
namespace eval nsB { ls *.ad} Ø invalid command name "ls" ❶
nsA::ls                       Ø invalid command name "nsA::ls" ❷
```

❶ Fails because `nsB` does not have an unknown handler.
❷ Fails because call is made from outside the `nsA` namespace context.

If no unknown command handler is set for a namespace, the global handler `::unknown` will be called instead (Section 3.5.1.1).

16.5.4. Introspecting name resolution: namespace which|origin

```
namespace origin NAME
namespace which ?-command? ?-variable? NAME
```

Tcl provides two commands `namespace which` and `namespace origin` which map names to their fully qualified versions.

The `namespace which` command returns the fully qualified version of *`NAME`* as per the name resolution rules. The switches `-command` (default) and `-variable` indicate whether *`NAME`* refers to a command or a variable respectively.

We will use a slightly modified version of the example in our previous section. We add an additional namespace `middleman` which imports and re-exports from `nsA`.

```
namespace eval nsA {
    proc aproc {} { puts "aproc called" }
    namespace export aproc
}
namespace eval middleman {
    namespace import ::nsA::aproc
    namespace export aproc
}
namespace eval importer { namespace import ::middleman::aproc }
namespace eval pathfinder { namespace path ::nsA }
```

Let us see how the command works with imported names and namespace paths.

```
namespace eval importer { namespace which -command aproc}   → ::importer::aproc
namespace eval pathfinder { namespace which -command aproc} → ::nsA::aproc
```

Notice that in the first instance, the returned fully qualified name is within the current namespace. This makes sense since the import of a name actually creates a command of that name in the importing namespace.

In the second instance, there was no command of that name created in the `pathfinder` namespace. Hence the namespace path of `pathfinder` is searched and the fully qualifed name of `aproc` is returned corresponding to the namespace in which it was found.

The `-variable` switch works similarly except that it follows the name resolution rules for variables.

```
namespace eval nsA {
    variable avar
    proc demo {} {
        variable avar
        namespace which -variable avar
    }
}
nsA::demo
→ ::nsA::avar
```

It suffices for the variable to have been **created**, it need not be **defined** (see Section 3.6.5.3 for the distinction).

For both commands and variables, the command returns the fully qualified name if found and an empty string otherwise.

While `namespace which` locates a command and returns the fully qualified path, `namespace origin` serves a different purpose. The fully qualified name it returns is that of **original** command even if there are “intermediate” namespaces importing and re-exporting the name. Contrast the two in our example:

```
namespace eval importer { namespace which -command aproc} → ::importer::aproc
namespace eval importer { namespace origin aproc}         → ::nsA::aproc
```

We see that `namespace which` returns the current namespace `importer` since the import of `aproc` resulted in the creation of a command of that name within the `importer` namespace. In contrast, `namespace origin` traverses through all intermediate links (`middleman` in our case) to the original command `nsA::aproc`.

The command will work with namespace paths as well.

```
% namespace eval pathfinder { namespace origin aproc}
→ ::nsA::aproc
```


The `namespace which` command can be used in lieu of `info commands` to check for the existence of a command. It is often preferred because unlike `info commands`, it does not treat its argument as a **pattern**. This makes it safer when checking existence of commands whose names may contain wildcard characters[1].

16.6. Namespace ensembles

In most languages, namespaces are limited to a single (albeit important) purpose — preventing conflicts between identifiers defined by independent modules. In Tcl, namespaces also provide the basis of another piece of useful functionality, *ensemble commands*.

16.6.1. Ensemble commands

An ensemble command is a command that has subcommands that collectively perform a set of related functions. We have already seen several examples within Tcl, such as `string` (Chapter 4) and `clock` (Chapter 11).

Namespaces offer a means to construct your own ensemble commands.

16.6.2. Creating ensembles: namespace ensemble create

```
namespace ensemble create ?OPTION VALUE?
```

Assume we want to encapsulate various operations related to Fibonacci sequences under the command `fib`. We will support three simple commands,

```
fibonacci sequence N
fibonacci nth N
fibonacci sum N
```

that return a sequence of length `N`, the `N` 'th number in the sequence and the sum of a sequence of length `N` respectively. We will make use of the `math` package in Tcllib[2] to do the hard work.

[1] Use of non-alphabetic characters in procedure names is not uncommon. For example, you will find `?` suffixes used for procedure names that return booleans, or `*` for extended forms of standard commands.
[2] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

```
package require math
namespace eval fib {
    proc nth {n} { return [math::fibonacci $n] }
    proc sequence {n} {
        set seq {}
        for {set i 1} {$i <= $n} {incr i} {
            lappend seq [nth $i]
        }
        return $seq
    }
    proc sum {n} {return [::tcl::mathop::+ {*}[sequence $n]]}
}
```

We can now call it using the standard namespace syntax

```
fib::sequence 3 → 1 1 2
fib::sum 3      → 4
```

To convert this to an ensemble command we need to make use of the `namespace ensemble create` command. For our example, this is very simple. By default, when no options are specified, `namespace ensemble create` will create an ensemble command of the same name as the namespace from which it is called. The subcommands will be the exported commands.

```
namespace eval fib {
    namespace export *
    namespace ensemble create
}                               → ::fib
fib nth 4                       → 3
fib sequence 3                  → 1 1 2
fib sum 5                       → 12
```

16.6.2.1. Naming an ensemble command

Readers who are at least half-awake will object that the name of the command is wrong; we wanted it to be `fibonacci`, not `fib`. The obvious way to fix this would be to change the name of the containing namespace itself to `fibonacci`. We will instead follow a different path of configuring the ensemble as that provides more flexibility in cases where the ensemble construction is not based on a single namespace. The `-command` option allows us to name the ensemble command.

```
% namespace eval fib {namespace ensemble create -command ::fibonacci}
→ ::fibonacci
% fibonacci nth 6
→ 8
```

Note the value we passed to the `-command` option was fully qualified. Otherwise we would have created the command `fibonacci` inside the `fib` namespace instead of at the global level as we wanted.

The `-command` option is also used to define an ensemble command without creating a new namespace as we will see in a later example.

The `namespace ensemble create` command accepts the options described below for `namespace ensemble configure` (Section 16.6.3).

16.6.3. Configuring ensembles

```
namespace ensemble configure COMMAND ?OPTION ?VALUE? …?
```

Having looked at the simplest method for creating ensembles, we will now look at configuration options that allow for more flexible construction of ensembles using the `namespace ensemble configure` command.

The *`COMMAND`* argument is the name of the ensemble command being configured. If no options are specified, the command returns the current values of the options.

```
% namespace ensemble configure ::fibonacci
→ -map {} -namespace ::fib -parameters {} -prefixes 1 -subcommands {} -unknown {}
```

If a single option is specified, with no associated value argument, the command returns the value of the option.

```
namespace ensemble configure ::fibonacci -namespace → ::fib
```

If multiple arguments are specified after the ensemble name *`COMMAND`*, they are interpreted as option and value pairs and the ensemble command is configured as per the specified values. An exception is `-namespace` which is a read-only option and cannot be modified.

16.6.3.1. Subcommand configuration: -subcommands, -map

In our example above, all commands exported from the `fib` namespace became subcommands of the command ensemble. If this is not desired, the `-subcommands` option lets you control exactly which subcommands are available through the ensemble. These need not be exported commands.

```
% namespace ensemble configure ::fibonacci -subcommands {nth sum}
```

Now only the two listed commands are callable through the ensemble.

```
% fibonacci nth 4
→ 3
% fibonacci sum 5
→ 12
% fibonacci sequence 3 ❶
Ø unknown or ambiguous subcommand "sequence": must be nth, or sum
```

❶ Error because `sequence` was not included in the `-subcommands` value

If the value of the `-subcommands` option is the empty list, which is the default, all commands exported from the namespace become ensemble subcommands. Resetting to the default makes `sequence` available again.

```
namespace ensemble configure ::fibonacci -subcommands {} → (empty)
fibonacci sequence 3                                     → 1 1 2
```

In contrast to the `-subcommands` option, `-map`, whose value should be a command prefix (Section 14.3.1.1), lets an ensemble subcommand be mapped to commands outside the namespace.

Consider the procedure `fib::nth` which does nothing other than call the `math::fibonacci` command from the `math` library. Instead of defining that procedure, we could have used the `-map` option to directly invoke `math::fibonacci`. Let us use this method to define a new subcommand `term` that does the same thing as `nth`. Additionally, making note that the mapping target is a **command prefix** and not just a command, we will define another subcommand `term4` which always returns the fourth number in the sequence.

```
namespace ensemble configure ::fibonacci -map {
    term ::math::fibonacci
    term4 {::math::fibonacci 4}
} -subcommands {term term4 sequence sum}
```

And of course, it all works as advertised.

```
fibonacci term 4 → 3
fibonacci term4  → 3
```

The `-map` and `-subcommands` options together control the subcommands available in an ensemble and the implementations to which they are mapped.

- If neither `-subcommands` nor `-map` is configured (or are empty) the ensemble subcommands are exactly those exported by the namespace.
- If the `-map` option is specified but `-subcommands` was an empty list (or unspecified), the ensemble commands are exactly the keys of the dictionary passed with `-map`.
- If `-subcommands` is specified (and not empty) the ensemble subcommands are exactly those listed in the option value. The corresponding implementation is that supplied in the `-map` dictionary argument if the subcommand is found there, or a procedure of the same name in the namespace linked to the ensemble.

16.6.3.2. Subcommand prefixes: option -prefixes

By default, ensemble commands accept unique prefixes for subcommands.

```
% fibonacci su 4
→ 7
```

This feature can be controlled with the `-prefixes` option. If you want exact matching of subcommands, you can disable the feature by setting the option to `false`.

```
% namespace ensemble configure ::fibonacci -prefixes false
% fibonacci su 4
Ø unknown subcommand "su": must be sequence, sum, term, or term4
```

16.6.3.3. Subcommand positioning: option -parameters

For additional flexibility, the position in which the subcommand appears in the ensemble command can be controlled with the `-parameters` option. For example, suppose we wanted to implement an ensemble command `arith` for simple arithmetic using infix notation. So instead of commands like

```
arith + 3 4
arith - 5 2
```

we would be able to write

```
arith 3 + 4
arith 5 - 2
```

Thus in effect we want the subcommands `+`, `-` to be positioned **after** the first argument to the command.

The first step is to define a simple namespace implementing the commands.

```
namespace eval arith {
    proc + {operand increment} {expr {$operand + $increment}}
    proc - {operand decrement} {expr {$operand - $decrement}}
}
```

Then we use the `-parameters` option to specify the parameters that appear **before** the subcommand in the ensemble. The option value is a list of elements corresponding to the number of arguments that should appear before the subcommand. The actual values of the list elements are only used to generate meaningful error messages and do not have any relevance otherwise.

For our example, we want a single argument before the subcommand.

```
namespace eval arith {
    namespace export + -
    namespace ensemble create -parameters {operand}
}
→ ::arith
```

We can now use infix notation for the ensemble.

```
arith 3 + 4 → 7
arith 5 - 2 → 3
arith       Ø wrong # args: should be "arith operand subcommand ?arg ...?" ❶
```

❶ Note use of `operand` in error message

16.6.4. Handling unknown subcommands: option -unknown

Just as for global commands and commands within a namespace, Tcl provides a means for an application to handle errors when an ensemble command is not defined. This is done with the `-unknown` ensemble configuration option.

If the ensemble's `-unknown` option has the default value of the empty string, any attempt to invoke a subcommand that is not defined will result in an error.

```
arith 2 * 3 Ø unknown or ambiguous subcommand "*": must be +, or -
```

If the `-unknown` option is configured for the ensemble and is not the empty string, it is registered as the unknown handler for subcommands for that ensemble and is called when a subcommand cannot be resolved as described in the previous sections. The entire attempted command is appended to the unknown handler before its invocation.

The return value from the unknown handler must be a valid (possibly empty) list. If the returned list is not empty, Tcl replaces the original ensemble command as well as the original subcommand with the words from the returned list and re-executes the replacement with the additional arguments from the original invocation.

An example will clarify how this works. Let us assume that for our `arith` ensemble command, if the subcommand has not been defined we will attempt to treat it as a standard operator defined in the `tcl::mathop` namespace. So we define the following unknown handler `delegator` for the ensemble.

```
namespace eval arith {
    proc delegator {args} {
        if {[llength $args] != 4} {
            error "Wrong number of arguments: should be \"[lindex $args 0] \
                operand operator operand\""
        }
        return ::tcl::mathop::[lindex $args 2]
    }
}
namespace ensemble configure ::arith -unknown ::arith::delegator
```

Now, if we try to execute a subcommand that has not been defined for `arith`, say `*`, `delegator` will be invoked with the name of the ensemble and all additional arguments. So when we execute

```
arith 2 * 3 → 6
```

`delegator` is invoke with arguments `::arith`, `2`, `*` and `3`. After some error checking, the command returns `::tcl::mathop::*`. Tcl then executes this command passing it the additional arguments `2` and `3` and returning the result.

If the unknown handler returns a list that is empty, Tcl will then attempt to run the **original** command again. What this does is to allow the unknown handler to add appropriate commands "on the fly".

Continuing with our example, considering going through the unknown handler for every invocation of `*` would be inefficient, we can instead add each subcommand the first time it is referenced.

Let us see how this might be implemented by redoing our example from scratch. This also illustrates that we can create an ensemble without an explicit namespace by using the global one in conjunction with the `-command` option.

```
namespace delete ::arith ❶
proc delegator {args} {
    if {[llength $args] != 4} {
        error "Wrong number of arguments: should be \"[lindex $args 0] operand \
            operator operand\""
    }
    lassign $args cmd - op
    set escaped_op [string map {* \\* ? \\? [ \\[ ] \\] \\ \\\\} $op]  ❷
    if {[llength [info commands ::tcl::mathop::$escaped_op]] == 0} {
        error "Invalid operator \"$op\""
    }
    set map [namespace ensemble configure $cmd -map]
    dict set map $op ::tcl::mathop::$op
    namespace ensemble configure $cmd -map $map
    return ""
}
namespace ensemble create -command arith -map {} -parameters {operand} -unknown \
    [namespace current]::delegator -prefixes false ❸
→ ::arith
```

❶ Get rid of our prior example
❷ The `string map` is used to escape operators that might also be interpreted as special characters by `info commands`
❸ Prefixes disabled so (for example) `=` will not be treated as a valid prefix of `==`.

The above code works as follows:

- The unknown handler `delegator` adds the operator to the command ensemble.
- Because `delegator` returns an empty string, Tcl will attempt to run the **original** command again.
- Since the operator has been added to the command ensemble, this attempt succeeds.
- Subsequent invocations of that operator will result in direct calls to the ensemble without the overhead of the unknown handler.

We have created the ensemble command as before but with **no subcommands defined**. Now everytime we invoke a new (valid) operator, it will be added as a subcommand. We can see this in the following sequence.

```
namespace ensemble configure arith -namespace → :: ❶
namespace ensemble configure arith -map       → (empty) ❷
arith 2 == 3                                  → 0
arith 2 * 3                                   → 6
arith 2 = 3                                   Ø Invalid operator "=" ❸
namespace ensemble configure arith -map       → == ::tcl::mathop::==
 * ::tcl::mathop::* ❹
```

❶ Notice linked namespace is the global namespace
❷ Initial subcommand map is empty
❸ `=` is not a valid operator
❹ The subcommand map is dynamically filled so `delegator` is called only once per operator

This method of updating subcommands on the fly is often seen in code that implements object systems based on namespaces where object method names are discovered dynamically[3].

16.6.5. Checking for ensembles: namespace ensemble exists

```
namespace ensemble exists COMMAND
```

The command `namespace ensemble exists` returns `1` if its argument is a ensemble command and `0` otherwise.

```
namespace ensemble exists string        → 1
namespace ensemble exists puts          → 0
namespace ensemble exists nosuchcommand → 0
```

16.6.6. Nested ensembles

Ensembles can be nested so that the command takes **multiple** subcommand arguments that form a hierarchy. Suppose there is a image manipulation package which implements a `image` command that handles PNG and JPEG image formats and some set of operations like resizing or rotating.

A possible interface it presents might look like

```
image png resize PNGDATA WIDTH HEIGHT
image jpeg rotate PNGDATA DEGREES
```

This interface is easy to create with nested namespaces as shown below.

[3] The Windows COM IDispatchEx interface being one example.

```
namespace eval image::png {
    proc rotate {imagedata degrees} {
        puts "Rotating PNG image"
    }
    proc resize {imagedata height width} {
        puts "Resizing PNG image"
    }
    namespace export *
    namespace ensemble create
}

namespace eval image::jpeg {
    proc rotate {imagedata degrees} {
        puts "Rotating JPEG image"
    }
    proc resize {imagedata height width} {
        puts "Resizing JPEG image"
    }
    namespace export *
    namespace ensemble create
}
namespace eval image {
    namespace export *
    namespace ensemble create
}
→ ::image
```

We can then call the commands in straightforward fashion.

```
% image png rotate "Some binary PNG" 90
→ Rotating PNG image
% image jpeg resize "Some binary JPEG" 640 480
→ Resizing JPEG image
```

16.6.7. Examples of ensembles

Enhancing existing commands

There are times when there is some commonly used functionality you wish was provided by a built-in command. Let us say you wanted the `encoding` command ensemble (Section 9.1) to include a new subcommand, `convertfromhex`, to decode an encoded byte sequence passed as a hexadecimal representation.

```
proc convertfromhex {args} {
    set hex [lpop args]
    return [encoding convertfrom {*}$args [binary decode hex $hex]]
}
set map [namespace ensemble configure ::encoding -map]
dict set map convertfromhex convertfromhex
namespace ensemble configure ::encoding -map $map
```

We can then call it as any other `encoding` subcommand.

```
encoding convertfromhex iso8859-1 4142e9 → ABé
```

There is one caveat though. Someone may have the same bright idea and define a new subcommand of the same name which has a different function. It is probably wise to use a prefix or some other means to prevent name clashes.

Indexing lists by name

Data records in Tcl are often stored as lists with individual fields accessed or set using `lindex` or `lset`. For example, a student record might be stored as a list containing the student name, age and college.

```
% set rec {Manute 18 {College of Engineering}}
→ Manute 18 {College of Engineering}
% puts "[lindex $rec 0] is [lindex $rec 1]."
→ Manute is 18.
```

Accessing fields using list indices can be error-prone, particularly when the number of fields is large. One can use dictionaries or write accessor functions to access the list. The former is not always under your control depending on the interface returning the data. The latter is tedious to do for every record "type" in the application.

A more convenient way would be to define a student record as a list of named fields, and then access fields using names instead of indices making it more convenient, readable and less error prone. For example,

```
% record student {name age college}
→ ::student
% student $rec age                ❶
→ 18
% set rec [student $rec age 19]   ❷
→ Manute 19 {College of Engineering}
```

❶ Retrieve a field
❷ Update a record

The mechanism should be generic so we can use it to define records with arbitrary fields. Let us see how to provide such a facility through a command, `record`, that will create an ensemble to access fields by name.

The implementation, shown below, first ensures that there are no conflicts with existing commands of the same name as the record being created. It then creates an ensemble command, in the namespace of the caller, of the given record name. The subcommands correspond to field names and map to an anonymous procedure (defined in the variable `accessor`) that returns or updates the appropriate field. Following a common Tcl idiom, the command acts as an getter or setter depending on the presence of an additional argument.

```
proc record {recname fields} {
    if {[uplevel 1 [list namespace which $recname]] ne ""} {
        error "can't create command '$recname': A command of that name already \
            exists."
    }

    set index -1
    set accessor [list ::apply {
        {index rec args}
        {
            if {[llength $args] == 0} {
                return [lindex $rec $index]
            }
            if {[llength $args] == 1} {
                return [lreplace $rec $index $index [lindex $args 0]]
            }
            error "Invalid number of arguments."
        }
    }]
    set map {}
    foreach field $fields {
        dict set map $field [linsert $accessor end [incr index]]
    }

    uplevel 1 [list namespace ensemble create -command $recname -map $map \
        -parameters rec]
}
```

Since the code is generic, we could of course define other record types as well.

```
% record automobile {manufacturer model color}
→ ::automobile
% automobile {Ferrari CaliforniaT red} color
→ red
```

Command objects using ensembles

Our final example illustrates implementation of a “command as an object” idiom. We will implement an ordered set that can be used as follows

```
set oset [ordered_set::new]; # Creates a new empty ordered set
$oset METHOD ?ARG ….?
$oset destroy;               # Destroys the ordered set
rename $oset "";             # Alternate means of destroying the set
```

An ordered set is ordered in that elements are preserved in the order that they are added. Tcl dictionaries are order preserving so our implementation is very simple. We keep a nested dictionary, the first level being indexed by an object identifier with the corresponding value

being the content of the corresponding set, also stored as a dictionary thanks to the order preserving properties.

```
namespace eval ordered_set {
    variable nextid 0
    variable sets {}

    proc add {id elem} {
        variable sets
        dict set sets $id $elem $elem
        return
    }

    proc remove {id elem} {
        variable sets
        if {[dict exists $sets $id $elem]} {
            dict unset sets $id $elem
        }
        return
    }

    proc contents {id} {
        variable sets
        return [dict keys [dict get $sets $id]]
    }

}
```

Now we just have to arrange for creating ordered sets and cleaning up when they are destroyed. The latter is easy, we just have to get rid of the data, so we show that first. The only non-obvious part is the additional `args` argument, the reason for which will soon be clear.

```
proc ordered_set::cleanup {id args} {
    variable sets
    dict unset sets $id
}
```

We are left with needing a means to create an ordered set command object which was more or less the whole point of this whole exercise.

We first generate a unique name for the command object using the namespace variable `nextid` as a identifier counter and initialize the corresponding content of the `sets` dictionary to empty.

We then create a map of ensemble subcommands that are mapped to a command prefix consisting of the corresponding procedure name with the identifier of the object being passed as the first argument. This map is then used to create an ensemble command whose name is the name of the object.

The last thing we have to deal with is object destruction. Any command can be deleted by using `rename` to rename it to the empty string. The same will also hold for our object command so we need to arrange for the related data to be cleaned up from the `sets`

dictionary when the command is deleted. We do this by setting a `trace` (Section 14.2) on the command object to invoke our `cleanup` procedure when it is deleted. The trace callback appends additional arguments (that we do not use) which is why `cleanup` definition had an unused `args` parameter. Note that we had also added a `destroy` subcommand to our map that does the same thing as syntactic sugar.

Here is the command object creation procedure.

```
proc ordered_set::new {} {
    variable nextid
    variable sets
    set objname "::oset#[incr nextid]"
    dict set sets $nextid [dict create]
    set map [dict create \
                 add [list add $nextid] \
                 contents [list contents $nextid] \
                 remove [list remove $nextid] \
                 destroy [list ::rename $objname ""]]  ❶

    namespace ensemble create -command $objname -map $map
    trace add command $objname delete [list [namespace current]::cleanup \
        $nextid]
    return $objname
}
```

❶ Note `rename` is fully qualified else it will default to the current namespace.

We can try out our ordered sets.

```
set oset [ordered_set::new] → ::oset#1
$oset add fee               → (empty)
$oset add fie               → (empty)
$oset add fo                → (empty)
$oset contents              → fee fie fo
$oset add fie               → (empty) ❶
$oset contents              → fee fie fo
$oset remove fee            → (empty)
$oset contents              → fie fo
$oset destroy               → (empty)
$oset contents              Ø invalid command name "::oset#1" ❷
```

❶ Duplicate element, preserves existing order
❷ Object has been destroyed

Of course, our toy implementation only illustrates some basic techniques. A real implementation would be generalized, support inheritance and other features. The Tcl Wiki has several examples of object-oriented programming systems built on top of namespaces before the introduction of TclOO in Tcl 8.6

17

Libraries and Packages

One of the basic principles of software development is to collect implementations of commonly useful and widely applicable functionality into *libraries* that can be shared amongst multiple applications. The simplest mechanism for implementing such a library is as a set of procedures in a file that is then sourced by any application that makes use of its functionality. In the case of a large library, this file could be a "main" file that optionally sources other files that implement parts of the library. This simple approach needs to be enhanced to provide a means of locating the library on the file system, loading on demand, versioning etc.

For historical reasons, Tcl includes multiple mechanisms for working with libraries:

- An index based system where procedure names are stored in an index file that is looked up and loaded on procedure invocation,
- Tcl *packages* which define a structure for versioning, locating and loading libraries implemented through multiple scripts and extensions
- Tcl *modules* that implement a simpler and more performant way to locate and load libraries implemented in a single file

We will describe the particulars pertaining to all three in this chapter.

17.1. The Tcl system library

Some of the Tcl core commands, for example `msgcat`, are themselves implemented as a library of Tcl scripts. The name of the directory where this system library resides is stored in the `tcl_library` global variable.

```
set tcl_library → c:/tcl/magic/lib/tcl9.0
```

The same information is also available via the `info library` command which simply returns the value of the `tcl_library` global variable.

```
info library → c:/tcl/magic/lib/tcl9.0
```

When Tcl starts up, it sets the value of the `tcl_library` variable by checking the following locations in order for library scripts:

- The directory specified by the `TCL_LIBRARY` environment variable if it exists and references an appropriate directory

- Directories relative to a default location that is defined when the Tcl executable was compiled. This may even be inside a ZipFS virtual file system as we describe in Chapter 25
- Directories relative to the location of the Tcl executable
- Directories relative to the current working directory

The above locations are checked in order and the first one that contains the expected library scripts is used to initialize `tcl_library`.

17.2. Loading libraries on demand: auto_load

```
auto_load COMMANDNAME
```

In Section 3.5.1.1 we described Tcl's default handling of unknown commands. One of the steps described there was how the unknown command handler uses the `auto_load` command to locate definitions of commands that are not currently known to the Tcl interpreter. We now look at `auto_load` in more detail.

When the default handler for unknown commands is called, it uses the `auto_load` command to try and locate the definition of the command that was invoked.

The command returns `1` if the command could be located and defined and `0` otherwise. It works by searching the list of directories stored in the global variable `auto_path` (Section 17.3.5) for files named `tclIndex`. Each `tclIndex` file contains an index that maps command names to the associated script for creating that command. Generally, this command creation script is simply a `source` (Section 3.14) command that executes a file in the directory containing the `tclIndex` file.

17.2.1. The tclIndex files: auto_mkindex

```
auto_mkindex DIR ?GLOBPAT …?
```

The `tclIndex` file is actually just a Tcl script that adds entries to an array `auto_index` that maps command names to the script to be executed to define that command. Here is a line from the `tclIndex` file for the Tcl system library (Section 17.1).

```
set auto_index(::tcl::history) [list ::tcl::Pkg::source [file join $dir \
    history.tcl]]
```

The very first time the `history` command is referenced, Tcl's unknown command handler invokes `auto_load` which will

- First check the `auto_index` global array for an entry for the `history` command. If found, it executes the corresponding script which presumably will define the required command.
- If no entry is found in the `auto_index` array, `auto_load` will evaluate all `tclIndex` files found in the directories listed in the `auto_path` global variable.
- The first step is then repeated except that if there is still no matching entry in the `auto_index` array, `auto_load` returns `0` indicating the command was not found.

A `tclIndex` file may be written and maintained manually by hand but is usually generated using the `auto_mkindex` command.

The command processes all files in the directory *`DIR`* that match any of the file name patterns *`GLOBPAT`*. These patterns are in the syntax used by the `glob` (Section 12.3.5) command.

If no *`GLOBPAT`* arguments are specified, the command defaults to `*.tcl`. A `tclIndex` file containing `auto_index` entries of the form we saw above is then written to the same directory.

If you have procedure names that contain special `glob` pattern characters such as `*`, the `auto_mkindex` command can get confused.

17.3. Packages

Tcl *packages* are one way of bundling a library of commands and procedures identified by a name and version. An application desiring to use the functionality provided can request loading of the package.

17.3.1. Naming packages

Packages are identified by their name, for example `http`. Package names may contain arbitrary characters although it is advisable to avoid special characters. Packages may contain the `::` namespace separator character sequence as well. Note however that these are **not** treated as namespace characters as packages have no direct correlation with namespaces.

17.3.2. Package versioning

Over time, new releases of a library contain feature enhancements, bug fixes and so on. Version numbers are used to distinguish these releases. A Tcl installation may contain multiple versions of a single package and applications can choose which version they wish to use.

17.3.2.1. Package version syntax

Version identifiers take the form of a sequence of decimal numbers generally separated by a `.` character. For example, `8`, `8.6` and `8.6.1000` are all valid version numbers. Version numbers in this form, using only `.` separators, are assigned to *stable* releases, i.e. releases that have been deemed ready for production use.

As a special case, a version number may contain the letters `a` or `b` in place of exactly one `.` separator; for example, `8.6a5`, `8.6b7`. These versions indicate *unstable* releases where functionality might change and which might not have undergone sufficient testing to be considered production ready. The `a` and `b` signify “alpha” and “beta” quality releases.

The leftmost number in a version identifier is the **major** version and the following number, if present, is the **minor** version. By convention, package releases are expected to follow certain norms with respect to major and minor versions:

- Packages are expected to maintain **backward** compatibility within a major version. Thus the `1.2` release of a package is expected to maintain compatibility with applications that make use of versions `1.0` or `1.1` of the package. There is no expectation of **forward** compatibility. Applications that work with version `1.3` may not work with `1.2`.

- Conversely, a change in major versions implies potentially incompatible changes in functionality. This is true in both the forward and backward directions. For example, applications that work with version `2.0` of a package will not necessarily work with either `1.0` or `3.0`.

When a package is loaded, the version requirements it must satisfy are specified using one forms shown in Table 17.1.

Table 17.1. Package version requirements syntax

Requirement	Description
MIN-MAX	This requirement specifies a range within which the version must reside. If *`MIN`* and *`MAX`* are equal, the version must also be the same. Otherwise, the version must be at least *`MIN`* and strictly less than *`MAX`*.
MIN-	The version must be at least *`MIN`*. There is no limit for the upper bound for the version.
MIN	The version must be at least *`MIN`*. The upper bound of the permissible range is the next higher major version relative to *`MIN`*.

17.3.2.2. Comparing package versions: package vcompare|vsatisfies

```
package vcompare VERA VERB
package vsatisfies VER REQ ?REQ …?
```

Version numbers follow a sequence where higher version numbers indicate a later release of a package. When comparing version numbers, the leftmost version numbers have higher significance and any missing version fields are treated as `0`. For example, `8.6.1` is a later version than `8.5.100` while `8.6` and `8.6.0` are equal.

If a version number includes `a` and `b` as separators, they are treated as an additional version component with values `-2` and `-1` respectively. For example, `8.6b22` is treated as version `8.6.-1.22` and therefore less than `8.6.0`. Consequently, versions marked alpha or beta are naturally deemed earlier than stable versions having the same major and minor levels.

The `package vcompare` command can be used to compare two version numbers.

The command returns `-1`, `0` or `1` depending on whether *`VERA`* is less than, equal, or greater than *`VERB`*.

```
package vcompare 8.6 8.6b22 → 1
```

The `package vsatisfies` command offers a more flexible method to check if a package satisfies certain version requirements.

The command returns `1` if *`VER`* meets at least one of the requirements stipulated by the *`REQ`* arguments which must be in one of the forms shown in Table 17.1 and illustrated below. Refer to the table for semantics of each form.

```
package vsatisfies 8.6.6 8.5-8.7 → 1
package vsatisfies 8.5 8.7       → 0
package vsatisfies 8.7 8.7       → 1
package vsatisfies 8.6 8-        → 1
package vsatisfies 9 8-          → 1
package vsatisfies 8.6 8         → 1
package vsatisfies 9 8           → 0
```

For example, Tcl version 8.6.1 had some bugs in its I/O implementation so to avoid this version while allowing any other Tcl with major version 8 or above, you could write

```
if {![package vsatisfies [info patchlevel] 8-8.6.1 8.6.2-]} {
    error "Unsupported version"
}
```

The above will error for `8.6.1` as the upper bound is not included in the permitted range.

17.3.3. Introspecting packages: package names | version | files

The `package names` command enumerates all packages that are known (not necessarily loaded) to the Tcl interpreter. However, getting the complete list requires Tcl to have already searched all its library directories at least once. This can be done by attempting to load a non-existent package.

```
% catch {package require nosuchpackage}
→ 1
% package names
→ rcs widgetPlus TclOO math::rationalfunctions tkBlend2d tie::std::dsource inte...
```

The `package versions` command will return the available versions of a package or an empty list if the package is not installed.

```
package versions mime → 1.7.2
```

The `package files` will retrieve the script files that were loaded for a package in the order they were sourced.

```
% package require msgcat
→ 1.7.1
% package file msgcat
→ C:/Tcl/magic/lib/tcl9/9.0/msgcat-1.7.1.tm
```

Note however that the command will not list files that used the auto-load mechanism (Section 17.2).

17.3.4. Installing packages

Tcl does not have a standardized method of installing packages.

If you are using a OS-provided distribution, other OS-supported Tcl packages can be installed in the same manner as Tcl, for example from within the Bash shell

```
sudo apt-get install PACKAGE
```

The Windows installer based distributions do not have a remote update capability at the time of writing. However, they bundle many commonly used packages within the distributions. These can be individually installed or uninstalled through the standard Windows Control Panel `Programs and Features` dialog's `Change` menu option.

In cases where the distribution does not include the package or has a different version of the package than that desired, follow the package's installation instructions. In many cases, installation consists of simply extracting the contents of a compressed archive into a directory that is included in the package search path stored in the `auto_path` global variable.

17.3.5. Searching for libraries

The `auto_load` (Section 17.2) and the `package require` (Section 17.3.6) commands search a list of directories, the *search path*, to locate `tclIndex` and `pkgIndex.tcl` files respectively. This directory list is given by the `auto_path` global variable.

When a Tcl interpreter is created, `auto_path` is initialized by concatenating the following:

- The value given by the `TCLLIBPATH` environment variable. This value is treated as the string representation of a Tcl list each element of which specifies a directory. If the `\` character is used as directory separator on Windows, it must be doubled as `\\` to avoid Tcl interpreting as a backslash escape sequence.
- The directory given by the `tcl_library` global variable.
- The parent directory of the directory in `tcl_library`
- The directories listed in the `tcl_pkgPath` global variable if it exists.

Applications are free to add directories (or even remove them though this is generally not recommended) to the search path by modifying `auto_path` appropriately.

Tcl examines all directories listed in `auto_path` for `pkgIndex.tcl` files and evaluates them. These files register package names and versions into a package index database as described in Section 17.3.9. This package database is then checked at the time of loading.

This package search description does not apply to module-based packages. The procedure for those is described in Section 17.5.2.

17.3.6. Loading packages: package require

```
package require NAME ?REQ …?_
package require -exact NAME VERSION
```

Before using the commands implemented by a package, the package must be loaded into the Tcl interpreter with `package require`.

In the first form of the command, the optional *`REQ`* arguments indicate version requirements using the syntax described in Section 17.3.2.2. The second form requires the package to be the exact version specified.

The package is loaded in the following sequence of steps:

- If the package *`NAME`* is already loaded into the interpreter and meets the specified version requirements, the command returns. If the loaded package version does not meet version specifications, the command will raise an error as multiple versions of a package cannot be loaded into a single interpreter.
- If the package is not already loaded, the command checks the internal package index database. If a suitable version is found in there, it loads the package by evaluating the associated script.
- If not found in the package index, Tcl further searches for it as described in Section 17.3.5. If no matches are found, the command raises an error. If one or more matches are found, the command loads the latest version present (modulo the stable/unstable attribute described below) into the interpreter.

On success, `package require` returns the version of the loaded package. Some examples:

```
package require http            → 2.10.0 ❶
package require http 2-         → 2.10.0 ❷
package require http 2.8        → 2.10.0 ❸
package require -exact http 2.8 Ø version conflict for package "http": have 2.10.0,
 need exactly 2.8 ❹
```

❶ Any version of the `http` package
❷ Version 2 or later (2.0, 2.1, 3.0 etc.)
❸ Version 2.8, 2.9... but not 3.x
❹ No version other than 2.8

In a previous section, we illustrated use of the `package vsatisfies` command to check we are running Tcl with major version 8 or higher except for 8.6.1. We could also do the check with the following command because the Tcl itself presents a package interface.

```
package require Tcl 8-8.6.1 8.6.2- → 9.0.1
```


The `package require` command loads the package only into the interpreter invoking the command. This means you can actually load multiple versions of a package as long as they are loaded into different interpreters. Use of multiple interpreters is described in Chapter 23.

17.3.6.1. Choosing stable versus unstable packages: package prefer

```
package prefer ?stable|latest?
```

There is one other consideration when Tcl selects a package to load when multiple versions of the package are present. As discussed in Section 17.3.2.1, package versions are distinguished between stable and latest, possibly unstable, versions. The latter embed the `a` or `b` characters to mark them as alpha or beta versions.

When Tcl selects a package to load in response to a `package require` command, treatment of unstable packages is affected by the *selection mode*. This mode may have the values `stable` and `latest`. In `latest` mode, the highest version available of the package is chosen irrespective of whether it is stable or not. In `stable` mode, the highest *stable* version of the package is chosen unless no stable versions are available in which case it falls back to the highest unstable version.

The `package prefer` retrieves or sets this package selection mode. If no arguments are specified, the command returns the current mode. If `latest` is specified as an argument, the mode is set accordingly. Passing `stable` as an argument is a no-op — the current mode is not changed. In both cases, the command returns the mode.

Stable releases of Tcl set the mode to `stable` at startup unless the `TCL_PKG_PREFER_LATEST` environment variable is set in which case the mode is initialized to `latest`. The value of `TCL_PKG_PREFER_LATEST` is immaterial. Alpha and beta releases of Tcl start with the mode as `latest` which cannot be subsequently changed.

17.3.7. Checking if a package is loaded: package present

```
package present ?-exact? NAME REQ ?REQ…?
```

The `package present` command returns the version of a package if it has been loaded and raises an error otherwise. The command works exactly like `package require` (Section 17.3.6) except that it will not load the package if it was not already present.

The following will either show a dialog or write to `stderr` depending on whether the Tk GUI package has been loaded.

```
if {[catch {package present Tk}]} {
    puts stderr "No GUI present"
} else {
    tk_messageBox -message "GUI present"
}
→ No GUI present
```

17.3.8. Registering packages: package ifneeded

```
package ifneeded NAME VER ?SCRIPT?
```

When Tcl searches for a package, it evaluates all `pkgIndex.tcl` files found in the package search path (Section 17.3.5). A `pkgIndex.tcl` file is just a normal Tcl script and may contain any Tcl commands but it's primary purpose is to register with Tcl the package name, version, and the script to load the package.

This registration is done with the `package ifneeded` command. When this command is evaluated, Tcl makes a note that version *`VER`* of package *`NAME`* may be loaded by evaluating *`SCRIPT`*. This information is used when an application requests a package to be loaded.

Below is the content of the `pkgIndex.tcl` file for a package `sequences` that we will implement in the next section.

```
package ifneeded sequences 1.0 [list source [file join $dir seq_arith.tcl]]
```

Before evaluating a `pkgIndex.tcl` file, Tcl sets the global variable `dir` to the path of the directory containing the `pkgIndex.tcl` file being evaluated. In our example we use this to load the main script for our package.

The `pkgIndex.tcl` can be as sophisticated as you need it to be. It is advisable to keep it relatively short however, as it is read and evaluated during Tcl's package search even when it is not the package being requested.

Tcl includes a command, `pkg_mkIndex`, that creates `pkgIndex.tcl` files and a related command, `pkg::create`, that you can use instead of manually creating the file. However, manual creation is simple (as we saw above) for simple packages and for more complex cases these commands are sufficiently lacking that their use is discouraged. We therefore do not describe them further.

If the *`SCRIPT`* argument is not provided, the command returns the script that was registered for loading the package.

```
% package ifneeded fileutil [package require fileutil]
→ source c:/tcl/magic/lib/tcllib2.0/fileutil/fileutil.tcl
```

A single `pkgIndex.tcl` file may contain multiple `package ifneeded` commands each registering a different package or even a different version of each package. You will find this or similar methods used in "package bundles" like Tcllib[1] composed of multiple packages.

17.3.9. Creating packages: package provide

```
package provide NAME ?VERSION?
```

A package consists of

- zero or more Tcl script files
- zero or more binary executables in the form of shared libraries
- zero or more data files such as images
- a `pkgIndex.tcl` file containing commands that tell Tcl the package name, version and the script to be evaluated to load the package.

[1] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

We will illustrate the process of package creation through an example. Our package, named `sequences`, will provide procedures, `arith_term` and `geom_term`, for calculating the n^{th} term of arithmetic and geometric sequences respectively. We will modularize this rather large package by breaking up the implementation into two files in a directory, also called `sequences` (though the directory name need not match the package name). The first, `seq_geom.tcl`, implements geometric sequences.

```
# seq_geom.tcl
namespace eval seq {
    proc geom_term {a r n} {
        return [expr {$a * $r**($n-1)}]
    }
}
```

The second, `seq_arith.tcl`, implements arithmetic sequences and also serves as the main script for the package.

```
# seq_arith.tcl
namespace eval seq {
    proc arith_term {a i n} {
        return [expr {$a + ($n-1)*$i}]
    }
}
source [file join [file dirname [info script]] seq_geom.tcl]
package provide sequences 1.0
```

Because `seq_arith.tcl` is the main script for the package, it includes the `package provide` command at the end which informs Tcl when the file is sourced that the `sequences` package version `1.0` is now loaded into the interpreter.

The *`NAME`* argument to the `package provide` command is the name of the package. When used to define or create a package, the *`VERSION`* argument must be supplied and is taken as the version of the package being provided. If *`VERSION`* is not specified, the command returns the version number of the package if it has already been previously provided, and an empty string otherwise.

We saw earlier the use of the `package present` command to check if a package has already been loaded into the interpreter. The `package provide` command without the *`VERSION`* argument is an alternate means of doing this.

```
if {[package provide Tk] eq ""} {
    puts "Package Tk is not loaded."
}
```

17.4. Shared library extensions

Tcl commands can be implemented natively in shared libraries referred to as Tcl extensions. These commands are made available in a Tcl interpreter by loading the extension. Usually the

package author will provide a suitable `pkgIndex.tcl` file so that the extension can be loaded with the usual `package require` command.

17.4.1. Loading extensions: load

```
info sharedlibextension
load ?-global? ?-lazy? ?--? PATH ?INITNAME? ?INTERP?
```

When no `pkgIndex.tcl` file is provided or when you are authoring the package and have to create the `pkgIndex.tcl` file, the `load` command is used to load the extension.

Here *`PATH`* specifies the file path of the shared library. *`INITNAME`* is used to construct the name of the function in the extension that is to be called to initialize it. *`INTERP`* specifies the name of the interpreter into which the extension is to be loaded. By default, the extension is loaded in the interpreter that executes the `load` command. This is only relevant in applications using multiple Tcl interpreters (Chapter 23).

After loading the shared library, Tcl calls an initialization function within it. The name of this function is constructed by appending either `_Init` or `_SafeInit` to *`INITNAME`* depending on whether the interpreter is a normal or a safe interpreter (Section 23.10). If the shared library does not export a function of the constructed name, the `load` command will fail.

Prior to Tcl 9, the initialization name was constructed by title-casing *`INITNAME`* before appending `_Init`. For example, if *`INITNAME`* was `myext`, the name of the initialization function would be `Myext_Init`.

If most cases, the initialization name need not be specified as it is by convention the same as the shared library extension base name. Thus an extension `myext.so` can be loaded simply as

```
load /path/to/myext.so
```

The `-global` and `-lazy` options are very rarely used and not discussed here. Refer to the Tcl documentation of `load` for details.

The extension used for shared libraries differs between platforms. The `info sharedlibextension` command returns the file extension used for shared libraries.

```
info sharedlibextension → .dll
```

That allows us to write our above example as

```
load myext[info sharedlibextension]
```

Note however that in most cases the file extension need not be specified as it will be automatically added. Moreover, the full path need not be specified if the shared library is present on the search path for shared libraries for that system. Nevertheless, it is good practice to specify the full path so as to prevent errors in case there are multiple files of that name present on the search path.

17.4.2. Enumerating loaded extensions: info loaded

```
info loaded ?INTERP? ?INITNAME?
```

The `info loaded` command will return information about loaded extensions.

`INTERP` identifies the interpreter of interest and defaults to the current interpreter. If the *`INITNAME`* argument is not passed, the command returns a list, each element of which is a pair containing a path to the loaded extension and its initialization function prefix. If *`INITNAME`* is passed, the command returns the path of the loaded extension whose initialization prefix is *`INITNAME`*.

```
% info loaded
→ {c:/tcl/magic/lib/twapi5.0.2/win32-x86_64/tcl9twapi502.dll Twapi} {C:/Users/a...
% info loaded {} Registry
→ c:/tcl/magic/lib/registry1.3/tcl9registry13.dll
```

17.5. Modules

Although the Tcl package mechanism is flexible, that flexibility has a cost associated with it. When locating packages, Tcl has to go through the package directory search path looking for `pkgIndex.tcl` files. These have to then be read and evaluated as Tcl scripts. In a large Tcl application that loads many packages, this can result in a noticeable delay at startup time, particularly when the application resides on a remote network location. Tcl modules provide an alternate scheme for libraries that mitigates these startup costs at the price of some flexibility.

Tcl modules incorporate two changes that reduces the time required for locating them:

- The module name and version are encoded into the file name itself and Tcl does not need to evaluate a file to retrieve this information as it does with `pkgIndex.tcl` files in the package case.
- The directory search path for modules is more limited.

Although implemented differently than the package form we discussed earlier, Tcl modules are used with many of the same commands. In particular,

- The `package names` command (Section 17.3.3) used for discovering available packages includes modules as well.
- Modules are loaded with `package require` (Section 17.3.6). When searching for a module, Tcl will give preference for a module based implementation of a package before a traditional one if both have the same version.
- The versioning syntax for modules is the same as discussed in Section 17.3.2.

For this reason, any reference to "packages" henceforth will include what we will term as "traditional" packages as well as modules. The rest of this section only describes the areas where modules differ from traditional packages.

17.5.1. Module file names

A Tcl module is stored as a **single** file containing a Tcl script. The name of the file must be the package name, followed by a `-` character, followed by the package version and an extension of `.tm`, for example `http-2.10.1.tm`. Specifically, it must match the regular expression

```
([_[:alpha:]][:_[:alnum:]]*)-([[:digit:]].*)\.tm
```

It is strongly suggested that module based packages not use upper case characters in the package name. Since the package names are mapped to file names, there is potential for confusion between file systems that distinguish character case and those that do not.

If the package name contains any `::` character sequences, they are treated specially during the module search process as detailed next.

17.5.2. Searching for modules

One important respect in which module based packages differ from the traditional packages is in how they are located. Module searches do not follow the process for auto-loading and traditional packages described in Section 17.3.5.

When searching for a module based package, Tcl first constructs a partial file path for the module. This is the same as the package name with one change — any `::` character sequences in the package name are replaced by the directory separator character. For example, a package name of `math::calculus` would translate to a file base name of `math/calculus-*`. The `*` allows for different versions of the package. This partial path is appended to each directory present in the module search path. For every file matching the constructed path, the equivalent of the following command is executed:

```
package ifneeded PACKAGENAME VERSION [list source MODULEPATH]
```

Here *`VERSION`* is extracted from the module file name and *`MODULEPATH`* is the path to the matching file.

This effectively registers the module in the package database. Requiring the package will result in evaluation of the corresponding `source` command.

The module search path: tcl::tm::path list

```
tcl::tm::path list ?DIR …?
```

The module search path is completely independent of the `auto_path` global variable used for traditional packages. It can be retrieved with the `tcl::tm::path list` command.

```
% tcl::tm::path list
→ C:/Tcl/magic/lib/tcl9/site-tcl C:/Tcl/magic/lib/tcl9/9.0
```

This is the list of directories that Tcl will examine when looking for a module.

Module search path initialization

The module search path is initialized by adding directories in the following order:

- Subdirectories of the form `tcl*MAJOR*/*MAJOR*.*MINOR*` under the parent directory of the path returned by `info library`.
- Subdirectories of the form `tcl*MAJOR*/*MAJOR*.*MINOR*` under the parent directory of the current process executable path as returned by `info executable`.
- The contents of the `TCL*MAJOR*_*MINOR*_TM_PATH` environment variables interpreted as directories separated by the `;` character on Windows and `:` on other platforms.
- The contents of the `TCL*MAJOR*.*MINOR*_TM_PATH` environment variables. These are interpreted as above but their use is discouraged as the `.` character in environment variables is not portable.

In all the above paths, `*MAJOR*` is the major Tcl version of the current interpreter and `*MINOR*` takes on the values less than or equal to the minor version of the current interpreter. So for example, for Tcl 8.6, `*MAJOR*` would be `8` and `*MINOR*` would take on values between `0` and `6`.

Adding directories to the module search path: tcl::tm::path add

```
tcl::tm::path add ?DIR …?
tcl::tm::roots PATHS
```

Directories can be added to the module search path with the `tcl::tm::path add` command. Each argument is added to the front of the search path in order with directories already present being ignored.

There is an important restriction enforced in the directories included in the search path. No directory in the search path may be an ancestor of another. For example, the following will raise an error:

```
% tcl::tm::path add /temp/foo
% tcl::tm::path list
→ /temp/foo C:/Tcl/magic/lib/tcl9/site-tcl C:/Tcl/magic/lib/tcl9/9.0
% tcl::tm::path add /temp ❶
Ø /temp is ancestor of existing module path /temp/foo.
% tcl::tm::path add /temp/foo/bar ❷
Ø /temp/foo/bar is subdirectory of existing module path /temp/foo.
```

❶ Attempt to add parent directory of existing entry.
❷ Attempt to add a subdirectory of existing entry.

An alternative means of adding directories to the search path is the `tcl::tm::roots` command which adds zero or more “roots” to the search path. The command takes a single argument containing a list of directories. For each element `*ROOT*` in this list, the command adds directories of the form:

- `*ROOT*/tcl*MAJOR*/site-tcl` where `*MAJOR*` is the major version of this Tcl interpreter
- One or more directories `*ROOT*/tcl*MAJOR*/*MAJOR*.*MINOR*` for every value of `*MINOR*` that is less than or equal to the minor version of this Tcl interpreter.

For example, after evaluating the command

```
% tcl::tm::roots [file join [file home] tcl]
```

the module search path will look as follows:

```
% print_list [tcl::tm::path list]
→ C:/Users/apnad/Documents/tcl/tcl9/site-tcl
  C:/Users/apnad/Documents/tcl/tcl9/9.0
  /temp/foo
...Additional lines omitted...
```

Removing directories from the module search path: tcl::tm::path remove

```
tcl::tm::path remove ?DIR …?
```

The `tcl::tm::path remove` command removes directories from the module search path. Arguments that do not exist in the search path are ignored.

```
tcl::tm::path remove /temp/foo → (empty)
```

17.5.3. Installing modules

Installation of packages implemented as modules is done in the same distribution-specific manner described for traditional packages in Section 17.3.4.

If the module is not included with the specific Tcl distribution, installation by hand is very simple as modules must by definition be implemented as a single file. This file simply needs to be copied to an appropriate directory in the module search path.

17.5.4. Creating modules

In the simplest case, where the package is implemented as a single Tcl script, creating a module just involves naming the file appropriately. When multiple scripts are involved, the files can be concatenated together. To create a module for our example package we can execute the following at the Unix shell prompt:

```
cat seq_geom.tcl seq_arith.tcl > sequences-1.0.tm
```

The equivalent on Windows would be

```
copy seq_geom.tcl+seq_arith.tcl > sequences-1.0.tm
```

The module file name reflects the package name and version and has the `.tm` extension.

When multiple files are coalesced into a module file, make sure to concatenate them in the correct order in cases where the scripts evaluate code at run time. In such cases, files defining the procedures or data must appear before scripts that invoke them during the loading process.

17.6. Packages versus modules

How does one make a choice between distributing script libraries as a traditional package versus a module? There are several considerations:

- Modules are easier to distribute as they are a single file. Packages need to be archived into `zip`, `tar.gz` or similar format and unarchived on the target.
- Modules are faster to load unless they embed shared libraries.
- Library scripts that have significant platform or version-specific components are easier shipped as packages as the `pkgIndex.tcl` file can load the appropriate pieces based on runtime information.
- Libraries that include executable binaries are better implemented as packages. Although modules can support binaries by copying embedded data to the file system, there are several drawbacks to this. Copying the shared library out to disk and reading it back incurs a load time performance hit. Some heuristic based virus scanners also flag this behaviour of writing code to disk and executing it as reflective of malware.

17.7. Multiplatform packaging: platform package

It is useful and convenient for the end user if the package can be installed in single directory, perhaps on the network, from where it can be loaded into Tcl interpreters running on differing architectures. For packages that are purely script-based, this is obviously not an issue. However, if your package includes shared libraries, support for multiple platforms from a single installation directory is a little trickier because the `pkgIndex.tcl` file for the package must load the appropriate shared library from the installation directory. The `platform` package addresses this requirement of a well-defined means of identifying the operating system and architecture of the host system. The package is part of the Tcl core distribution but must be explicitly loaded before its commands can be invoked.

```
package require platform → 1.0.19
```

The package implements three commands, `identify`, `generic` and `patterns`, all of which are placed under the `platform` namespace.

The `platform::identify` command returns an identifier for the platform that encodes the operating system, C runtime version and CPU architecture. For example, on an Linux Ubuntu system,

```
% platform::identify
→ linux-glibc2.19-x86_64
```

while on a Windows 32-bit system,

```
% platform::identify
→ win32-ix86
```

The `platform::generic` command is similar except it returns a more generic identifier that encodes the "family" of platforms. For example, on the same Ubuntu system,

```
% platform::generic
→ linux-x86_64
```

The third command in the package is `platform::patterns`. This command takes an argument that is a platform identifier as returned by `platform::identify`. It then returns a list of all platform identifiers that are compatible with the one passed.

Again, on our Ubuntu system,

```
% platform::patterns [platform::identify]
→ linux-glibc2.19-x86_64 linux-glibc2.18-x86_64 linux-glibc2.17-x86_64 linux-gl...
```

A multi-platform package that includes binary extensions can store the shared library for a platform under a subdirectory named after the identifier returned by `platform::identify` or `platform::generic` for that platform. The `pkgIndex.tcl` script can then load the shared library by invoking `platform::identify` or `platform::generic` at runtime to locate the appropriate subdirectory.

17.7.1. The platform::shell package

```
platform::shell::identify SHELL
platform::shell::generic SHELL
platform::shell::platform SHELL
```

The `platform::shell` package is similar to the `platform` package except that while the latter returns platform information for the currently executing Tcl shell, the former returns information about a different Tcl shell residing on the same machine. The shell of interest is identified by its path and must be executable on the same machine as the current shell.

The package provides `platform::shell::identify` and `platform::shell::generic` commands that are functionally similar to the `platform::identify` and `platform::generic` commands except that they return the corresponding information for the targeted Tcl shell whose path is given by *`SHELL`*.

As an example, both 32-bit and 64-bit shells may be installed on the same 64-bit Windows machine. Assuming we are currently running the 64-bit shell,

```
package require platform                                    → 1.0.19
package require platform::shell                             → 1.1.4
platform::identify                                          → win32-x86_64
platform::shell::identify c:/tcl/9.0.0/x86/bin/tclsh90.exe → win32-ix86
```

The package also has an additional command `platform::shell::platform` which returns the contents of `platform` element of the target shell's `tcl_platform` array we described in Chapter 2.

```
platform::shell::platform c:/tcl/9.0.0/x86/bin/tclsh90.exe → windows
```

Where might one make use of the `package::shell` commands? The primary reason for the existence of these commands is for code repositories and installers where the Tcl shell running the installer script is not necessarily the same as the target shell for which a package is being installed. They allow the installer to detect the architecture of the target shell and copy the appropriate files there.

17.8. Introspecting package configuration

Packages (using the term generically to include modules and shared libraries) may wish to expose certain configuration information to applications such as implemented features, build information etc. They should do so by implementing a `pkgconfig` command within the package's namespace. This command should support at least the subcommands `list` and `get`. The `pkgconfig list` command should take no arguments and return a list of configuration keys. The `pkgconfig get` command should take a single argument which is a configuration key and return the corresponding value. Applications can then invoke this command to retrieve information about the package. Note that the key names and contents are entirely up to the package.

As an example, Tcl itself exposes its configuration through the `pkgconfig` command in `tcl` namespace.

```
% tcl::pkgconfig list ❶
→ debug threaded profiled 64bit optimized mem_debug compile_debug compile_stats...
% tcl::pkgconfig get optimized ❷
→ 1
```

❶ Lists all available keys provided by the Tcl package
❷ Tells us whether compiled with optimization enabled

18

Object-Oriented Programming

This chapter describes Tcl features that support object oriented programming. It does *not* go into detail about what constitutes object oriented programming, what its benefits are, or how your classes should be designed. The answers generally depend on who you ask and there have been enough words written on the topic.

Nevertheless, as we go along we will briefly describe some basic concepts for the benefit of the reader who really is completely unexposed to OO programming.

The OO programming facilities, TclOO, described in this book are those built into Tcl since version 8.6. There are other object-oriented implementations for Tcl in wide use, in particular *XoTcl* and its successor *nx*, and *Snit* (Snit's Not Incr Tcl) which is commonly used to implement Tk GUI widgets. These are not described here as they are not part of the Tcl core.

Most examples in this chapter are based on a framework for modeling banks. Our bank has accounts of different types, such as savings and checking. Permitted operations, like deposits and withdrawals, depend on the account type. Moreover, while we have certain privileged customers who get special treatment, we also have to follow certain directives from Big Brother. No, Bank of America cannot run its operations based on our framework, but it suffices for our illustrative purposes.

18.1. Objects and classes

The core of OO programming involves, no surprise, *objects*. An object, often a representation of some real world entity, captures state (data) and behaviour which is the object's response to *messages* sent to it. In most languages, implementation of these messages involves calling *methods* which are just function calls with a context holding the object's state. The state of an object representing a bank account would include the current balance and messages would deposit or withdraw funds.

A *class* is a template that defines the state and methods, collectively called members, encapsulated by objects of a specific type. More often than not, creating an object of the class, often known as instantiating an object, is one of the duties of the class. The terms *object* and *class instance* are used interchangeably.

Not every OO system has, or needs, the notion of a class. *Prototype* based systems instead create objects by *cloning* an existing object — the prototype — and defining or modifying members. TclOO provides facilities to support both the classy and the classless[1] models.

[1] No value judgement intended.

TclOO exposes sufficient internal functionality to allow you to layer your own object-oriented programming model on top. For most folks though, who are not into experimentation with OO as a way of life, the base functionality provided by TclOO more than suffices.

18.2. Class basics

18.2.1. Creating a class

```
oo::class create CLASS ?DEFINITION?
oo::define CLASS DEFINITION
oo::define CLASS DEFINITIONCMD ?ARG …?
```

The `oo::class create` command creates a TclOO class named *`CLASS`*. *`DEFINITION`* is an optional *class definition script* that specifies the state and behavior (methods) associated with objects of that class. Unlike in static languages, **the complete definition of the class does not have to be specified at the time of class creation**. Everything that defines a class can be subsequently modified with the `oo::define` command at runtime.

The first syntactic form of `oo::define` accepts the same *`DEFINITION`* syntax as the `oo::class create` command and which we describe in Section 18.2.2. In the second form, *`DEFINITIONCMD`* is any command that may be used in a class definition script.

So, for example, the fragment below builds a `Demo` class definition in piecemeal fashion.

```
oo::class create Demo {
    variable var
    method m0 {} {}
} ❶
oo::define Demo {
    method m1 {} {}
    method m2 {} {}
} ❷
oo::define Demo method m3 {} {} ❸
```

❶ Create a class.
❷ Modify the class using a definition script.
❸ Modify the class with a definition command.

We will use this runtime configurability of classes to incrementally build our application.

The class `Demo` is actually just another Tcl command and could have been created in any namespace, not necessarily the global one. Either of the following

```
oo::class create ns::Demo
namespace eval ns {oo::class create Demo}
```

would create a new class `Demo` in the `ns` namespace, entirely unrelated to our `Demo` class in the global namespace.

18.2.2. Class definition script

A class definition script as passed to `oo::class create` or `oo::define` is just a Tcl script that is executed in a special namespace scope which contains the commands shown in Table 18.1. Given it is a script, standard Tcl commands can also be used but are rarely needed.

Table 18.1. Class definition commands

Command	Description
`classmethod`	Defines a class method that runs in the namespace context of the containing class (Section 18.2.5.5).
`constructor`	Specifies the initialization of newly created objects of that class (Section 18.2.5.1).
`definitionnamespace`	Changes the namespace scope for the class and object definition scripts (Section 18.11.1).
`deletemethod`	Removes a method from all instances of the class (Section 18.2.5.6)
`destructor`	Specifies the script to run within the namespace context of an object at the time the object is destroyed (Section 18.2.5.1).
`export`	Makes an instance method of the class visible outside the object's context (Section 18.2.5.3).
`filter`	Updates the list of filter methods of objects belonging to the class (Section 18.8).
`forward`	Delegates an instance method implementation of the class to another command (Section 18.7).
`initialize`	Evaluates a script in object namespace of the class itself (Section 18.2.8).
`method`	Defines an instance method that will be run in the namespace contexts of objects of the class (Section 18.2.5.2).
`mixin`	Updates the list of classes that are mixed-in to objects of the class being defined (Section 18.6).
`private`	Evaluates a definition script in a namespace context that will be private to the class (Section 18.4.4).
`renamemethod`	Renames an instance method of the class (Section 18.2.5.7).
`self`	Returns the name of the class or evaluates an **object** definition script on the class.
`superclass`	Updates the list of superclasses for the class (Section 18.4).
`unexport`	Makes an instance method not callable from outside objects of the class (Section 18.2.5.3).
`variable`	Creates instance variables in the namespace contexts of all objects of the class (Section 18.2.4.1).

Let us now create the `Account` class which forms the basis of our application. We will incrementally enhance it as we move on through the chapter.

```
% oo::class create Account {}
→ ::Account
```

18.2.3. Destroying classes

A class is also an object and can be destroyed by invoking its `destroy` method.

```
% Demo destroy
```

Classes can also be destroyed by renaming to the empty string.

```
rename Demo ""
```

Destroying a class will

- delete its definition,
- destroy any classes that inherit from (Section 18.4), or mix-in (Section 18.6), the class,
- destroy all objects belonging to all destroyed classes.

18.2.4. Data members

Data members of a class fall into two categories.

- **Instance variables**, generally referred to simply as variables, hold data specific to each instance or object of the class. Updating an instance variable in one object does not affect the variable of the same name in another object.
- **Class variables** are shared between instances and hold data that is common to all instances of a class.

18.2.4.1. Instance variables: variable, my variable

```
variable ?SLOTOPERATION? ?VARNAME …?
my variable ?SLOTOPERATION? ?VARNAME …?
```

In our simple example, the state for an account object includes an account number and the current balance in the account. These are obviously data members that hold information specific to each account and therefore we define them as instance variables with the `variable` command within a class definition script.

There can be multiple `variable` statements within a definition script, each defining one or more data members. The optional *`SLOTOPERATION`* argument is detailed in Section 18.2.6 but in a nutshell it controls how the variable names defined in the command are combined with

those from previous `variable` commands. By default, the *`VARNAME`* arguments are added to the existing variable definitions.

Continuing to build up our example, we declare the data members for the class, `AccountNumber` and `Balance`. These are then visible within all methods of the class and can be referenced there **without any qualifiers or declarations**.

```
% oo::define Account {
    variable AccountNumber Balance ❶
}
```

❶ The author uses title case for data members to avoid inadvertent conflicts with names of arguments and local variables.

Data members do not have to be declared using `variable` in a class definition script. They can also be brought into scope within a method with the `my variable` command.

Note the difference between the `variable` command in a class definition script and the `variable` command in a namespace. The former only creates variables without initializing their values whereas the latter both creates and optionally initializes variables in a namespace.

18.2.4.2. Class variables: classvariable

```
classvariable VAR ?VAR …?
```

A class variable is a data member that is shared across all instances of a class. It is created and initialized within the script passed to the `initialize` command (Section 18.2.8) as part of a class definition. The `classvariable` command is then used to bring it into scope within a method context.

In our banking example, it is reasonable to assume the bank routing number is common to all accounts and need not be maintained separately for each. We can therefore choose to make it a class variable.

```
oo::define Account {
    initialize {
        variable RoutingNumber "0123456789" ❶
    }
    method getRoutingNumber {} {
        classvariable RoutingNumber
        return $RoutingNumber
    }
}
```

❶ Note `variable` in this context is the command to create namespace variables, not the one to create instance variables as described in the prior section. See the sidebar below.

Under the covers

As described in Section 18.3.4, every object is associated with a private namespace. Classes are themselves objects and therefore have their own private namespace. Class variables for the class are created within this namespace and not the namespace for each object instance. The `initialize` method always runs within context of the class namespace when an object is constructed and therefore in the example above, `RoutingNumber` is defined within that context.

When a method is invoked on an object, the class variable, defined in the class namespace, has to be brought within the scope of the method, running in the object namespace. The `classvariable` does exactly that, linking a local variable of the same name to the variable in the class namespace.

18.2.5. Methods

18.2.5.1. Constructors and destructors

Before we can start banking operations we need some means to initialize an `Account` object when it is created and clean up when it is destroyed. These tasks are performed through special methods named `constructor` and `destructor` both of which are optional. These differ from normal methods in two respects:

- They are not explicitly invoked by name. Rather, the `constructor` method is automatically run when an object is created. Conversely, the `destructor` method is run when the object is destroyed.
- The `destructor` method definition differs from other methods in that it does not have a parameter corresponding to arguments.

For our simple example, these methods are straightforward.

```
oo::define Account {
    constructor {account_no} {
        puts "Reading account data for $account_no from database"
        set AccountNumber $account_no ❶
        set Balance 10000
    }
    destructor {  ❷
        puts "[self] saving account data to database"
    }
}
```

❶ `AccountNumber` and `Balance` are instance variables available in all methods because of their creation using `variable` in the class definition script.

❷ Syntax of the `destructor` definition only takes the script argument.

18.2.5.2. Defining methods: method

```
method NAME ?VISIBILITY? ARGLIST BODY
```

Having defined the data members, let us move on to defining the methods that comprise the behaviour of an `Account` object. Methods are defined through the `method` command which, like `variable`, is executed inside a class definition script.

Other than the *VISIBILITY* parameter, discussion of which we defer to Section 18.2.5.3, a method is defined in exactly the same manner as `proc` defines a Tcl procedure. The difference with respect to a procedure lies in how it is invoked and the context in which the method executes.

Below we define some basic methods for account operations.

```
oo::define Account {
    method UpdateBalance {change} {
        set Balance [tcl::mathop::+ $Balance $change]
        return $Balance
    }
    method withdraw {amount} {
        return [my UpdateBalance -$amount]
    }
    method deposit {amount} {
        return [my UpdateBalance $amount]
    }
}
oo::define Account method balance {} { return $Balance } ❶
```

❶ Alternate syntax of `oo::define` for illustrative purposes.

18.2.5.3. Method visibility

Separation of public interfaces from private ones is a fundamental principle in OO systems. In TclOO, this separation is accomplished through the *visibility* of methods. The visibility, which may be *exported*, *unexported* or *private*, dictates the contexts in which a method can be invoked.

- Exported methods may be invoked from all contexts.
- Unexported methods may only be invoked from the containing object context.
- Private methods may only be invoked from within the context of a method defined in the same class as the private method. We will defer private methods to Section 18.4.4 after we discuss inheritance.

By default, methods beginning with a lower case letter are exported and unexported otherwise. Thus in our example, `deposit` and `withdraw` are exported while `UpdateBalance` is not. The visibility of a method can be explicitly specified at definition time by passing the *`VISIBILITY`* argument to `method` as `-export`, `-unexport` or `-private`.

Method visibility can also be changed subsquent to definition with the `export` and `unexport` commands in a class definition script. Thus

```
oo::define Account export UpdateBalance
```

would result in the unexported `UpdateBalance` method being exported. Conversely,

```
oo::define Account unexport UpdateBalance
```

will remove it from the exported list.

The `export` and `unexport` commands are not limited to the methods defined in that particular class. They can also be applied to methods inherited from superclasses or mix-ins, for example when a derived class wants to limit functionality supported by a base class.

18.2.5.4. The unknown method

Every object has a method named `unknown` which is run when a invoked method is not found in the method chain (Section 18.9) for that object. Its definition should take the form

```
method unknown {target_method args} {.. implementation ..}
```

The `unknown` method is passed the name of the invoked method as its first argument followed by the arguments from the invocation call.

The default implementation of this method, which is inherited by all objects from the root `oo::object` object, raises an error. Classes and objects can override the default implementation method to take some other action instead.

An example of its use is seen in the COM client implementation in the TWAPI Windows extension. The properties and methods exported from a COM component are not always known beforehand and in fact can be dynamically modified. The TclOO based wrapper for COM objects defines an `unknown` method that looks up method names supported by a COM component the first time a method is invoked. If found, the lookup returns an index into a function table that can then be invoked through the `ComCall` method. The implementation of `unknown` looks like

```
oo::define COMWrapper {
    method unknown {method_name args} {
        set method_index [COMlookup $method_name]
        if {$method_index < 0} {
            error "Method $method_name not found."
        }
        return [my ComCall $method_index {*}$args]
    }
}
```

This is a greatly simplified, not entirely accurate, description for illustrative purposes.

18.2.5.5. Class methods: classmethod, myclass

Analogous to class variables are class methods which run within the namespace context of the defining **class** as opposed to the defining **class instance** (object). In the case of inheritance, the context will be that of the derived class if the method is invoked on the derived class or an instance of the derived class.

The `classmethod` and `myclass` commands are not available in Tcl 8.6 and earlier.

By their very nature, class methods do not have access to any instance variables (which instance would it use?) but do have access to class variables as these are shared across all instances of the class. However, since class methods run in the namespace of the defining class and not that of a class instance, they use the standard `my variable` syntax to being the class variable into scope.

A class method to change the routing number for all accounts would look as follows:

```
oo::define Account {
    classmethod setRoutingNumber {newNumber} {
        my variable RoutingNumber
        set RoutingNumber $newNumber
    }
}
```

The class method can be used on either an instance of the class or the class itself.

```
% Account setRoutingNumber "1234567890" ❶
→ 1234567890
% Account create dummy "999999999"
→ Reading account data for 999999999 from database
  ::dummy
% dummy setRoutingNumber  "1234567890" ❷
→ 1234567890
```

❶ Invoke directly on the class
❷ Invoke via an object

Note however that there is a difference in the (inadvisible) case where there is an instance method of the same name as the class method. Invoking the method on the class will invoke the class method while invoking the method on the object will invoke the instance method. Within the context of an object method, `my` will also call the instance method. Use `myclass` command to call a class method in such cases.

Here is a macabre example to illustrate the difference between `my` and `myclass`.

```
oo::class create C {
    method suicide {} {my destroy}
    method genocide {} {myclass destroy}
}
```

Calling `suicide` only destroys the target instance while `genocide` destroys the class and all instances.

```
% C create c1; C create c2; C create c3; info class instances C
→ ::c1 ::c2 ::c3
% c1 suicide
% info class instances C
→ ::c2 ::c3
% c2 genocide
% info class instances C
Ø C does not refer to an object
```

18.2.5.6. Deleting methods: deletemethod

```
deletemethod METHOD ?METHOD …?
```

Method definitions can be deleted at any time with the `deletemethod` command inside a class definition script.

```
% oo::class create C {method print args {puts $args}}
→ ::C
% C create c
→ ::c
% c print some nonsense
→ some nonsense
% oo::define C {deletemethod print}
% c print more of the same
Ø unknown method "print": must be destroy
```

Deletion of methods from classes is rarely used. However, deletion of methods from objects is sometimes useful in object specialization (Section 18.5).

18.2.5.7. Renaming methods: renamemethod

```
renamemethod FROMNAME TONAME
```

The `renamemethod` command is used within a class definition script to rename an existing method. Once renamed, the method must be invoked by its new name, even for existing objects.

```
% oo::class create C {method print args {puts $args}}
→ ::C
% C create c
→ ::c
% oo::define C renamemethod print output
% c print foo
Ø unknown method "print": must be destroy or output
% c output foo
→ foo
```

We have not discussed class inheritance as yet, but we will just note here that renaming a method in a class will not rename methods of the same name in any ancestors or descendents for that class.

18.2.5.8. Method callbacks: callback, mymethod

```
callback METHODNAME ?ARG …?
mymethod METHODNAME ?ARG …?
```

Many Tcl commands accept a callback that will be executed later, such as event handlers and **lsort** comparators. Methods on objects can also serve as callbacks, but they require the correct object context to be established when the method is invoked. The `callback` command, which must be used **within a method call**, generates the appropriate wrapper to set up the context for method *`METHODNAME`* to be invoked as the callback. The result of `callback` can then be invoked in any context and sets up the appropriate object context for *`METHODNAME`*. The additional *`ARG`* arguments will be passed as the initial arguments to *`METHODNAME`* when it is called.

In the example below, `callback_demo` expects to invoke an arbitrary command prefix. Since it will execute in a global context, the method passed to it needs to be wrapped so that it executes in the object's context when invoked as a callback.

```
proc callback_demo {callback} {
    puts "Callback: \"$callback\""
    {*}$callback C D E
}
oo::class create CallbackExample {
    method print_args {args} {
        my variable Var
        puts "print_args: [join $args ,],$Var"
    }
    method invoke_callback {} {
        my variable Var
        set Var X
        callback_demo [callback print_args A B]
    }
}
CallbackExample create cbObj
cbObj invoke_callback
→ Callback: "::oo::Obj965::my print_args A B"
  print_args: A,B,C,D,E,X
```

The `mymethod` command is an alias for `callback`.

The `callback` and `mymethod` commands are not available in Tcl 8.6 and earlier.

18.2.5.9. Methods as commands

```
link { COMMANDNAME ?METHODNAME?} …
```

Methods are associated with a particular object and have to be invoked either via that object name or through the `my` command within that object's context. The `link` command arranges for a method on an object to be invoked just as any other Tcl command without any reference to the object name or the `my` command. The command must be called within an object's context and creates a command *`COMMANDNAME`* relative to the object's namespace context if it is not fully qualified. Invocation of the command will result in a call to the method named *`METHODNAME`* on the object in whose context the `link` command was invoked. If *`METHODNAME`* is not specified, it defaults to *`COMMANDNAME`*.

See Section 18.11.3 for an example.

18.2.6. Slot operations

TclOO uses the term **slot** as a generic term for superclasses, variables, mix-ins and filters. While these are all completely different in their purpose and operation, they all have one commonality in that a class or an object is associated with a **list** of each. These lists share certain characteristics:

- They can be configured at runtime through commands `superclass`, `variable` etc. to add or remove elements.
- The order of elements in the list has semantic consequences, for example in method lookups in the presence of multiple superclasses.

This last point implies that there needs to be a mechanism for controlling the order of elements when elements are added to the list. Each command defines a default behavior that we note in the discussion of each command. For example, the `filter` command **appends** its arguments to the existing filters. On the other hand, `superclass` **replaces** existing superclasses. However, in many instances finer control is desired and this is what is provided by the **slot operations** shown in Table 18.2 that can be passed to the relevant commands.

Table 18.2. TclOO slot operations

Slot operation	Description
`-append`	Appends the arguments to the current slot contents.
`-appendifnew`	Appends the arguments to the current slot contents if they are not already present.
`-clear`	Sets the slot contents to an empty list. No arguments should be supplied to the command.
`-prepend`	Prepends the arguments to the current slot contents.
`-remove`	Removes the arguments from the current slot contents if they are present.
`-set`	Replaces the slot content with the passed arguments.

For example, the following append, prepend and remove a filter method `Log` in a class.

```
filter Log
filter -prepend Log
filter -remove Log
```

18.2.7. Modifying an existing class

As we have seen in previous sections, you can incrementally modify a class using `oo::define`. Practically nothing about a class is sacred — you can add or delete methods, data members, change superclasses or mix-ins, and so on.

The question then arises as to what happens to objects that have already been created if a class is modified. The answer is that the changes in the class are also reflected in existing objects so for example any new methods can be invoked on them. Or if you add a mix-in or a superclass, the method lookup sequence for the object will be appropriately modified.

However, some care should be taken when modifying a class since existing objects may not hold all state expected by the new class. For example, the new constructors are (obviously) not run for the existing objects and thus some data members may be uninitialized. The modified class implementation has to account for such cases.

18.2.8. Class initializer: initialize

```
initialize INITSCRIPT
```

The class initializer is a script that is called at the time of class definition to perform any additional setup of the namespace of the class object itself (Section 18.3.4). The `initialize`, alternately spelt as `initialise`, command runs this script within a class definition script.

We saw an example of its use in Section 18.2.4.2.

Do not confuse the constructor with the initializer. The former initializes objects (instances) of the class when they are created. The latter runs at class definition time in the namespace of the class object itself.

18.3. Working with objects

Having defined our model, we can now begin operation of our bank.

18.3.1. Creating an object: OBJECT create|new

```
CLASS create OBJNAME ?ARG …?
CLASS new ?ARG …?
```

An object of a class is created by invoking one of two built-in methods on the class itself. The `create` method creates an object with a specific name. The `new` method generates a name for the created object.

Creating an object also initializes it, passing the arguments to the constructor.

```
% set acct [Account new 3-14159265]
→ Reading account data for 3-14159265 from database
  ::oo::Obj966
% Account create smith_account 2-71828182
→ Reading account data for 2-71828182 from database
  ::smith_account
```

The created objects are Tcl commands and as such can be created in any namespace.

```
% namespace eval my_ns {Account create my_account 1-11111111}
→ Reading account data for 1-11111111 from database
  ::my_ns::my_account
% Account create my_ns::another_account 2-22222222
→ Reading account data for 2-22222222 from database
  ::my_ns::another_account
```

Note that `my_account` and `my_ns::my_account` are two distinct objects.

18.3.2. Destroying objects

Objects in Tcl are **not** garbage collected as in some other languages and have to be explicitly destroyed by calling their built-in `destroy` method. This also runs the object's `destructor` (Section 18.2.5.1) method. Any operation on a destroyed object will naturally result in an error.

```
% my_ns::my_account destroy
→ ::my_ns::my_account saving account data to database
% my_ns::my_account balance
Ø invalid command name "my_ns::my_account"
```

Objects are also destroyed when its class or containing namespace is destroyed.

```
% namespace delete my_ns
→ ::my_ns::another_account saving account data to database
% my_ns::another_account balance
Ø invalid command name "my_ns::another_account"
```

18.3.3. Invoking methods

```
OBJECT METHODNAME args….
```

Methods can be invoked on objects in a manner similar to ensemble commands with the method name passed as the first argument.

```
$acct balance          → 10000
$acct deposit 1000     → 11000
$acct getRoutingNumber → 1234567890
```

When calling a method from another method **in the same object context**, the alias `my` is used to refer to the current object. So the `deposit` method we saw earlier in Section 18.2.5.2 calls the `UpdateBalance` method as

```
my UpdateBalance $amount
```

Class methods can also invoked on objects in the same manner. Earlier we invoked the `setRoutingNumber` class method on the `Account` class itself. We can also invoke it on an object of that class.

```
$acct setRoutingNumber "1234567890" → 1234567890
```

18.3.4. Namespace contexts

Every object has a unique namespace associated with it. This namespace holds the instance variables for the object. Methods run in the context of their object's namespace. This means the object data members such as `Balance`, defined through `variable`, are in the scope of the method and can directly be referenced without any qualifiers as seen in the method definition earlier. We will introspect object namespaces further in Section 18.12.5.

The namespace context also makes available several commands — such as `self`, `next` and `my` — which can only be called from within a method. We have already seen the use of `my` to invoke other methods in an object. Its function is to set up the appropriate namespace context so the called method knows on which object it is being invoked. We will see further uses of `my` and the other commands as we go along.

18.3.5. External access to data members: my varname

```
my varname VARNAME
```

Data members are not directly accessible from outside the object. Methods, such as `balance` in our example, have to be defined to allow callers to read and modify their values. While this may be desirable from a software engineering perspective, there are some cases where a instance or class variable needs to be accessed from outside the object or class. The `my varname` command can be used to retrieve the fully qualified name of an instance variable.

The following example illustrates two such cases where a fully qualified name is needed.

```
oo::class create Alarm {
    variable Signal
    constructor {delay} {
        after $delay [list set [my varname Signal] 1]
        vwait [my varname Signal]
    }
}
Alarm new 1000
Alarm destroy
```

- The `after` (Section 19.3) command schedules the `set` command to run in the global namespace after a delay. Because it runs in the global namespace, the `set` needs to be passed the fully qualified name of the instance variable `Signal`.
- The `vwait` (Section 19.2.3.1) command expects the variable name passed to it to be relative to the global namespace.

While the `my varname` needs to be invoked from within a method, it is in fact also possible to get the fully qualified name of an instance variable from other contexts using TclOO introspection features. See Section 18.12.5.

18.4. Inheritance: superclass

```
superclass ?SLOTOPERATION? ?SUPERCLASS …?
```

The defining characteristic of OO systems is support for *inheritance*.

Inheritance allows a *derived* class (also refered to as a *subclass*) to inherit the behavior and other attributes of another class — called its *base* class or *superclass* — with a view to then extend or otherwise modify it. In Tcl, subclasses inherit from a superclass using the `superclass` command within the subclass' class definition script.

The class within whose definition the command is invoked will then be set up to inherit from the list of specified classes. If the class already inherits from any superclasses, they will be replaced. This behavior can be changed by passing the optional *SLOTOPERATION* argument as described in Section 18.2.6.

In our banking example, we may define separate classes representing savings accounts and checking accounts, each inheriting from the base account and therefore having a balance and methods for deposits and withdrawal. Each may have additional functionality, for example check writing facilities for the checking account and interest payments for the savings account.

The intention behind inheritance is to model *is-a* relationships. Thus a checking account *is a* bank account and can be used at any place in the banking model where the behaviour associated with a bank account is expected. This *is-a* relation is key when deciding whether to use inheritance or some other facility such as mix-ins.

Let us define our `SavingsAccount` and `CheckingAccount`. Instead of using `oo::define` as before, we will provide the full class definition as part of the `oo::class` command itself.

First, a class for the checking account,

```
oo::class create CheckingAccount {
    superclass Account ❶
    method cash_check {payee amount} {
        my withdraw $amount
        puts "Writing a check to $payee for $amount"
    }
}
```

❶ A `CheckingAccount` is also an `Account` and inherits its behavior.

and similarly, a class for the savings account.

```
oo::class create SavingsAccount {
    superclass Account
    variable MaxPerMonthWithdrawals WithdrawalsThisMonth
    constructor {account_no {max_withdrawals_per_month 3}} {
        next $account_no
        set MaxPerMonthWithdrawals $max_withdrawals_per_month
    }
    method monthly_update {} {
        my variable Balance
        my deposit [my MonthlyInterest]
        set WithdrawalsThisMonth 0
    }
    method withdraw {amount} {
        if {[incr WithdrawalsThisMonth] > $MaxPerMonthWithdrawals} {
            error "You are only allowed $MaxPerMonthWithdrawals withdrawals a \
                month"
        }
        next $amount
    }
    method MonthlyInterest {} {
        my variable Balance
        return [format %.2f [tcl::mathop::* $Balance 0.005]]
    }
}
```

The `superclass` command in the class definition establishes that `SavingsAccount` and `CheckingAccount` inherit from `Account`. This statement by itself means they will behave exactly like the `Account` class, with the same methods and variables defined. Further commands in the definition script will extend or modify the class behaviour.

18.4.1. Methods in derived classes

Methods available in the base class are available in derived classes as well. In addition, new methods can be defined, such as `cash_check` and `monthly_update` in our example, that are only present on objects of the derived class.

If the derived class defines a method of the same name as a method in the base class, it overrides the latter and will be called when the method is invoked on an object of the derived class. Thus the `withdraw` method of the `SavingsAccount` class overrides the `withdraw` method of the base `Account` class.

18.4.1.1. Chaining methods

```
next ?ARG …?
nextto CLASSNAME ?ARG …?
```

In the `withdraw` method, we are just modifying the original method's functionality with an additional condition, not replacing it. Therefore, after making the check we want to just pass on the request to the base class method and not duplicate its code. This is done with the `next`

command which invokes the superclass or mixin method with the same name as the current method. The `nextto` command is similar except it allows control of the superclass whose method is to be invoked in the case of multiple inheritance. Both `next` and `nextto` may be called at any point in a method, not necessarily in the beginning or the end.

Constructors and destructors are also chained. If a derived class does not define a constructor, as is true for the `CheckingAccount` class, the base class constructor is invoked when the object is created. If the derived class does define a constructor, that is invoked instead and it is up to that constructor to call the base class constructor using `next` as appropriate. Destructors behave in a similar fashion.

This *method chaining* is actually only an example of a broader mechanism we will explore in detail in Section 18.9.

18.4.2. Data members in derived classes

Derived classes can define new data members using either `variable` in the class definition or `my variable` within a method as in `withdraw`.

Because data members are **always** defined in the namespace of the object, you have to careful about conflicts between variables of the same name being defined in a base class and a derived class if they are intended to represent different values. Use *private contexts* (Section 18.4.4) to protect against inadvertent conflicts.

Data members defined in a parent (or ancestor) class are also accessible within a derived class **but** they have to be brought within the scope of the method through the `variable` declaration in the derived class definition or the `my variable` statement within a method as is done in the implementation of `MonthlyInterest`. Although we use a direct variable reference there for expository purposes, in the interest of data hiding and encapsulation, direct reference to variables defined in ancestors should be avoided if possible. It would have been better to write the statement as

```
my deposit [format %.2f [* [my balance] $rate]]
```

Let us try out our new account types by creating one of each.

```
% SavingsAccount create savings S-12345678 2
→ Reading account data for S-12345678 from database
  ::savings
% CheckingAccount create checking C-12345678
→ Reading account data for C-12345678 from database
  ::checking
```

The `SavingsAccount` adds one method, `monthly_update` and overrides `withdraw`.

```
savings withdraw 1000  → 9000
savings withdraw 1000  → 8000
savings withdraw 1000  Ø You are only allowed 2 withdrawals a month
savings monthly_update → 0
```

Checking facilities are available only for checking accounts.

```
% checking cash_check Payee 500
→ Writing a check to Payee for 500
% savings cash_check Payee 500
Ø unknown method "cash_check": must be balance, deposit, destroy, getRoutingNum...
```

18.4.3. Multiple inheritance

Imagine our bank also provides brokerage services. Accordingly we define a new class.

```
oo::class create BrokerageAccount {
    superclass Account
    method buy {ticker number_of_shares} {
        puts "Buying $number_of_shares shares of $ticker"
    }
    method sell {ticker number_of_shares} {
        puts "Selling $number_of_shares shares of $ticker"
    }
}
```

The company now decides to make it even more convenient for customers to lose money in the stock market. So we come up with a new type of account, a Cash Management Account (CMA), combining the features of the checking and brokerage accounts. We can model this in our system using *multiple inheritance*, where a class inherits from more than one parent.

```
oo::class create CashManagementAccount {
    superclass CheckingAccount BrokerageAccount
}
```

Be careful when using multiple `superclass` statements as the earlier declarations are overwritten if the `-append` option is not specified. The above example written using multiple `superclass` commands would be written as:

```
oo::class create CashManagementAccount {
    superclass CheckingAccount
    superclass -append BrokerageAccount
}
```

Our CMA account can do it all.

```
% CashManagementAccount create cma CMA-00000001
→ Reading account data for CMA-00000001 from database
  ::cma
% cma cash_check Payee 500
→ Writing a check to Payee for 500
% cma buy GOOG 100
→ Buying 100 shares of GOOG
```

Use of multiple inheritance is a somewhat controversial topic in OO circles. Be as it may, TclOO offers the facility, as well as an alternative using mix-ins (Section 18.6). Programmers can then make the design choices they deem appropriate.

18.4.4. Private contexts

We have so far discussed methods that were either exported, which made them callable from all contexts, or not exported which made them callable only from within an object's context. The latter includes **all** methods defined for that object including those in subclasses, mixins, filters etc. This broke isolation between class definitions.

- An author of a subclass had to be careful when defining a method (even unexported) to not give it the same name as a method in the superclass unless the intent was to override it.
- At the same time, if the superclass implementation changed, necessitating the addition of a new method for private use, there is the risk that it will be hidden by an existing method of the same name in a subclass.

Since the two classes may be written by different authors or teams, and the superclass may not be even aware of the existence of the subclass that uses it, both are real risks.

Suppose the `Account` class had been defined with a requirement of a minimum balance of 1000 after a withdrawal and this requirement was enforced by a `CheckMinimum` method invoked from the `withdraw` method.

```
oo::define Account {
    method CheckMinimum {withdrawal} {
        if {($Balance-$withdrawal) < 1000} {
            error "Withdrawal failure: minimum balance not met."
        }
    }
    method withdraw {amount} {
        my CheckMinimum $amount
        return [my UpdateBalance -$amount]
    }
}
```

Now we can continue to withdraw small amounts but larger amounts fail.

```
% cma withdraw 100
→ 9400
% cma withdraw [cma balance]
Ø Withdrawal failure: minimum balance not met.
```

Now, independent of the banking group above, at some point the brokerage group decides that trading being a risky business, purchase of shares requires a minimum of 5000 in the account. Accordingly, a check is added to the `buy` method in `BrokerageAccount` and the programmer tasked with implementing the new policy coincidentally chooses the same method name `CheckMinimum` for checking minimum balances.

```
oo::define BrokerageAccount {
    method CheckMinimum {cost_of_shares} {
        if {([my balance]-$cost_of_shares) < 5000} {
            error "Cannot buy shares: minimum balance not met."
        }
    }
    method buy {ticker number_of_shares} {
        my CheckMinimum [expr {$number_of_shares * 10}]  ❶
        puts "Buying $number_of_shares shares of $ticker"
    }
}
```

❶ Assume we get 10/share cost from somewhere

We are now rightly stopped from purchasing shares **but we cannot withdraw money either** though we meet the minimum balance requirements for withdrawals.

```
% cma balance
→ 9400
% cma buy TCL 500
Ø Cannot buy shares: minimum balance not met.
% cma withdraw 5000
Ø Cannot buy shares: minimum balance not met.
```

The error message should make the problem apparent — the `BrokerageAccount.CheckMinimum` method is inadvertently overriding the `Account.CheckMinimum` method even though the method `withdrawal` is defined in `Account` and not `BrokerageAccount`. The exported / unexported categorization does not provide enough granularity of method scope.

Private contexts specifically target this issue. Definitions in a private context are only visible in the class they are defined and are always preferred over inherited, derived, and mixed-in ones when referenced within instances of their defining class, thereby avoiding inadvertent overrides.

Private contexts are not available Tcl 8.6 and earlier.

18.4.4.1. Defining private contexts: private

```
private CMD ?ARG …?
private SCRIPT
```

The `private` command inside a class or object definition sets up a private context for methods, forwarded methods and variables.

In the first form, *`CMD`* must be `method`, `forward`, `self` or `variable` and subsequent argument syntax is that for the corresponding command. *`CMD`* is defined within the private context for the class.

In the second form, *SCRIPT* can be any class or object definition script. Any `method`, `forward`, `self` or `variable` commands within the script will be defined in the private context.

18.4.4.2. Private methods and forwards

In the case of methods and forwards, definition in a private context has two effects.

- The method or forward can only be invoked within methods defined within that same class, or same instance in case of private contexts defined within `oo::objdefine` definition scripts. In particular, private methods cannot be called even from subclasses.
- The method or forward is placed at the front of the method chain (Section 18.9) so any references to its name from within other methods defined in the same class will always select the private method.

We will redefine the `CheckMinimum` method in `Account` as private to resolve the conflict we created earlier.

```
oo::define Account {
    private method CheckMinimum {withdrawal} {
        if {($Balance-$withdrawal) < 1000} {
            error "Withdrawal failure: minimum balance not met."
        }
    }
}
```

We can verify that we are still prevented from purchasing shares but are now allowed to withdraw money as expected. The `withdraw` method in `Account` is calling the `CheckMinimum` within `Account` and not in the derived class.

```
% cma balance
→ 9400
% cma buy TCL 500
Ø Cannot buy shares: minimum balance not met.
% cma withdraw 5000
→ 4400
```

We could also have defined `CheckMinimum` using either of the alternative syntaxes shown below — the script argument to `private`, or passing `-private` to the `method` command.

```
oo::define Account {
    private {
        method CheckMinimum {withdrawal} {...}
        ... other private methods and variables ...
    }
}
oo::define Account {
    method -private CheckMinimum {withdrawal} {....}
}
```

Unlike unexported methods, private methods are not restricted to be called only via the `my` command. They can be invoked on an object from another object belonging to the same class.

The example below illustrates the difference.

```
oo::class create C {
    method unexportedMethod -unexport {} {
        puts "unexportedMethod called"
    }
    method privateMethod -private {} {
        puts "privateMethod called"
    }
    method doCall {otherObject methodName} {
        $otherObject $methodName
    }
}
```

If we try to invoke the unexported method on `secondObject` from the context of `firstObject`, the call fails. A call to a private method however succeeds but only because it is made from a method within the context of an instance of the same class.

```
% C create firstObject
→ ::firstObject
% C create secondObject
→ ::secondObject
% firstObject doCall secondObject unexportedMethod
Ø unknown method "unexportedMethod": must be destroy, doCall or privateMethod
% firstObject doCall secondObject privateMethod
→ privateMethod called
```

18.4.4.3. Private variables

In a similar fashion to private methods, private variables defined within a class are not accessible from subclasses as illustrated below.

```
% oo::class create PrivateVarDemo {
    variable var
    private variable privateVar
    constructor {} {set var "non-private data"; set privateVar "private data"}
    method getPrivateVar {} { return $privateVar }
    method getfqn {} { return [my varname privateVar] }
}
→ ::PrivateVarDemo
% oo::class create PrivateVarDemoSubclass {
    superclass PrivateVarDemo
    method subGetVar {} {my variable var; return $var}
    method subGetPrivateVar {} {my variable privateVar; return $privateVar} ❶
}
→ ::PrivateVarDemoSubclass
```

❶ Will fail because subclasses cannot access variables defined in the private context of a superclass.

If we try out the class, we can see that while the private variable can be accessed by the class where it is defined, the subclass can only access the non-private `var` variable and an attempt to access the private variable raises an error.

```
PrivateVarDemoSubclass create demo → ::demo
demo getPrivateVar                 → private data
demo subGetVar                     → non-private data
demo subGetPrivateVar              Ø can't read "privateVar": no such variable
```

A private variable actually creates a hidden variable with a generated name which is then linked to a variable with the specified name only within the context of that class. We can see this with the `varname` command that returns the fully qualified name of a member variable.

```
demo getfqn       → ::oo::Obj992::988 : privateVar
set [demo getfqn] → private data
```

18.5. Specializing objects: oo::objdefine

```
oo::objdefine OBJ DEFINITION
oo::objdefine OBJ DEFINITIONCMD ARG ?ARG …?
```

The next thing we talk about, object specialization, may be new to readers who are more familiar with class-based OO languages such as C++ where an object's methods are exactly those defined for the class(es) to which the object belongs.

In TclOO on the other hand, we can further "specialize" an *individual* object by overriding, hiding, and deleting methods defined in the class or even adding new ones. In fact, the potential specialization includes features such as forwarding, filters and mix-ins but we leave those for now as we have not discussed them as yet. As we will see, we can even change an object's class.

Specialization is done through the `oo::objdefine` command which is analogous to the `oo::define` command for classes except that it takes an object as its argument instead of a class. With the obvious exception of the commands `constructor`, `destructor` and `superclass`, all commands, like `method`, `variable`, `export` etc., available within `oo::define` scripts can also be used inside the script passed to `oo::objdefine`. The difference is that they act on a specific class instance (object) instead of a class.

Like `oo::define`, `oo::objdefine` also has two syntactic forms. The *`DEFINITION`* argument is an *object definition script* as described in the following section while *`DEFINITIONCMD`* is one of the commands that may be used in the definition script.

18.5.1. Object definition script

The object definition script passed to `oo::objdefine` is very similar to the class description scripts described in Section 18.2.2 except that certain commands that do not make sense within the context of an object are not available.

Table 18.3 shows a summary of the object definition commands. These commands operate in very similar fashion in a object definition script as they do in a class definition script. We do not further describe them separately as for the most part the only difference is that they impact the specific object and not all objects of a class.

Table 18.3. Object definition commands

Command	Description
`deletemethod`	Deletes a method for the object (Section 18.2.5.6).
`export`	Makes an instance method visible outside the object's context (Section 18.2.5.3).
`filter`	Updates the list of filter methods for calls made to instance methods of the object (Section 18.8).
`forward`	Delegates an instance method of the object to another command (Section 18.7).
`method`	Defines an instance method for the object (Section 18.2.5.2).
`mixin`	Updates the list of classes that are mixed-in to the object (Section 18.6).
`private`	Evaluates a definition script in a namespace context that will be private to the object (Section 18.4.4).
`renamemethod`	Renames a method for the object (Section 18.2.5.7).
`self`	Returns the name of the object.
`unexport`	Makes an instance method not callable from outside the object (Section 18.2.5.3).
`variable`	Creates instance variables in the namespace context of the object (Section 18.2.4.1).

18.5.2. Object-specific methods

Let us illustrate object specialization with our banking example. Imagine our banking system had the requirement that individual accounts can be frozen based on an order from the tax authorities. We need to define a procedure we can call to freeze an account so all transactions on the account will be denied. Correspondingly, we need a way to unfreeze an frozen account.

```
proc freeze {account_obj} {
    oo::objdefine $account_obj {
        method UpdateBalance {args} {
            error "Account is frozen. Don't mess with the IRS, dude!"
        }
        method unfreeze {} {
            oo::objdefine [self] { deletemethod UpdateBalance unfreeze }
        }
    }
}
```

When the `freeze` procedure is passed an `Account` object, it uses `oo::objdefine` to override the `UpdateBalance` method that was part of the object's class definition with a object specific `UpdateBalance` method that raises an error instead.

It then defines a new method `unfreeze` that can be called on the object at the appropriate time to restore things back to normal. We could have actually defined an `unfreeze` procedure instead of a `unfreeze` method as follows:

```
proc unfreeze {account_obj} {
    oo::objdefine $account_obj deletemethod UpdateBalance
}
```

This would have accomplished the same job in a clearer manner. We chose to implement an `unfreeze` method instead to illustrate that we can actually change an object's definition even from **within** the object.

There are a couple of points that need to be elaborated:

- The `self` command is only usable within a method and returns the name of the current object when called without parameters. Thus the `oo::objdefine` command is instructed to modify the object itself.
- Although not required in our example, it should be noted that variables defined in the class are not automatically visible in object-specific methods. They need to be brought into scope with the `my variable` command.
- When called from within a `oo::objdefine` script, the `deletemethod` erases the specified object-specific methods. It does **not** affect methods defined in the class so the original `UpdateBalance` will still be in place and will no longer be overridden.

Let us see how all this works. At present Mr. Smith can freely withdraw money.

```
smith_account withdraw 100 → 9900
```

So far so good. Now we get a court order to freeze Mr. Smith's account. Next time Mr. Smith tries to withdraw some money, a lesson is learnt.

```
% freeze smith_account
% smith_account withdraw 100
Ø Account is frozen. Don't mess with the IRS, dude!
```

Have we affected other customers? No.

```
$acct withdraw 100 → 10900
```

Cornered Mr. Smith pays up to unfreeze the account.

```
smith_account unfreeze     → (empty)
smith_account withdraw 100 → 9800
```

Notice that the class definition of `UpdateBalance` was not lost in the process of adding and deleting the object-specific method.

This ability to define object-specific methods can be very useful. Imagine writing a computer game where the characters are modeled as objects. Several characteristics of the objects, such as the physics determining movement, are common and can be encapsulated with a class definition. The special "powers" of each character cannot be part of this class and defining a separate class for each character is tedious overkill. The special power of a character can instead be added to the character's object as a object-specific method. Even modeling scenarios like temporary loss of a power without a whole lot of conditionals and bookkeeping becomes very simple using the object specialization mechanisms.

18.5.3. Changing an object's class

Being a true dynamic OO language, TclOO can even change the class of an object through `oo::objdefine`. One could change a savings account to a checking account.

```
% set acct [SavingsAccount new C-12345678]
→ Reading account data for C-12345678 from database
  ::oo::Obj993
% $acct monthly_update
→ 0
```

So far so good. Let us attempt to cash a check.

```
% $acct cash_check Payee 100
Ø unknown method "cash_check": must be balance, deposit, destroy, getRoutingNum...
```

Naturally that fails because it is not a checking account. Not a problem, we can fix that by morphing the object to a `CheckingAccount`.

```
oo::objdefine $acct class CheckingAccount → (empty)
```

We can now cash checks successfully

```
$acct cash_check Payee 100 → Writing a check to Payee for 100
```

but monthly updates no longer work as the account is no longer a `SavingsAccount`.

```
% $acct monthly_update
Ø unknown method "monthly_update": must be balance, cash_check, deposit, destro...
% $acct destroy
→ ::oo::Obj993 saving account data to database
```

Needless to say, you have to be careful when "morphing" objects in this fashion since data members may differ between the two classes.

Note the optional form of the oo::objdefine command that we have used in the above code fragment.

While morphing of accounts might seem contrived, consider the use of the feature in a state machine wherein each state is represented by a class implementing the state's behavior. State changes would be implemented by the state machine object changing its class to the class corresponding to the new state.

18.6. Using mix-ins

Earlier we looked at the use of inheritance to extend a class. We will now look at another mechanism to extend or change the behaviour of classes (and objects) — mix-ins.

The literature on the subject describes mix-ins in several different ways, often depending on language-specific capabilities. From this author's perspective, a mix-in packages a bundle of related functionality such that it can be used to extend one or more classes or objects. In some languages, multiple inheritance is used for this purpose but we will postpone that discussion until after we have seen an example of a mix-in.

Going back to our banking model, imagine we have an Electronic Fund Transfer (EFT) facility that provides for transferring funds to other accounts. We will not worry about how this is done but just assume global procedures are available for the purpose. This facility is available to all checking accounts but only to selected savings accounts. There are several ways to implement this but our preference in this case is for mix-ins over the alternatives for reasons we discuss later.

18.6.1. Adding a mix-in to a class: mixin

```
mixin ?SLOTOPERATION? ?MIXINCLASS …?
```

The specified *MIXINCLASS* classes are installed as mix-ins to the class within whose definition `mixin` is invoked. By default, any existing mix-in classes are replaced but this behavior can be changed by passing `SLOTOPERATION` as described in Section 18.2.6.

A mix-in is defined as a class in exactly the same manner as we have seen earlier. In fact, in theory any class can be a mix-in. What sets a mix-in apart is the conceptual model and how the class is **used**. In our example, the EFT facility would be modeled as a class that implements two methods, `transfer_in` and `transfer_out`. Conceptually, the class does not represent an *object*, but rather a *capability* or, as is termed in some literature, a *role*. It adds functionality to a "real" object.

```
oo::class create EFT {
    method transfer_in {from_account amount} {
        puts "Pretending $amount received from $from_account"
        my deposit $amount
    }
    method transfer_out {to_account amount} {
        my withdraw $amount
        puts "Pretending $amount sent to $to_account"
    }
}
```

Since we want all checking accounts to have this facility, we will add `EFT` to the `CheckingAccount` class as a mix-in. This is accomplished with the `mixin` command within a class definition script.

We can add the EFT facility to `CheckingAccount` as a mix-in and thereby enable it for all checking accounts.

```
% oo::define CheckingAccount mixin EFT
% checking transfer_out 0-12345678 100
→ Pretending 100 sent to 0-12345678
% checking balance
→ 9400
```


Modifying the class definition in **any** manner, in this case adding a mix-in, also impacts **existing** objects of that class. Thus the `checking` object automatically supports the new functionality.

In the case of savings accounts, we only want **select** accounts to have this facility. Assuming our `savings` object represents one of these privileged accounts, we can add the mix-in to just that object through `oo::objdefine`.

```
% oo::objdefine savings {mixin EFT}
% savings transfer_in 0-12345678 100
→ Pretending 100 received from 0-12345678
  8140.0
% savings balance
→ 8140.0
```

Notice that the `EFT` class does not really know anything about accounts. It encapsulates features that can be added to **any** class or object that defines the methods `deposit` and `withdraw` required to support the mix-in's functionality. So if we had a `BrokerageAccount` class or object, we could mix it in there as well.

18.6.2. Using multiple mix-ins

A class or object may have multiple classes mixed in. So for example if we had a facility for electronic bill presentment implemented as a mix-in class `BillPay`, we could have added it along with `EFT` as a mix-in in a single statement

```
oo::define CheckingAccount {mixin BillPay EFT}
```

or as multiple statements

```
oo::define CheckingAccount {
    mixin EFT
    mixin -append BillPay
}
```

By default, the `mixin` command overwrites existing mix-in configuration so in the absence of the `-append` option when using multiple `mixin` statments, only class `BillPay` would be mixed into `CheckingAccount`.

18.6.3. Mix-ins versus inheritance

Because one of its goal is to provide the required infrastructure for additional OO systems to be built on top, TclOO offers a wide variety of capabilities that sometimes overlap in their effect. The question then arises as to how to choose the appropriate feature for a particular design requirement. One of these design choices involves mix-ins and inheritance.

We offer the author's thoughts on the matter. Luckily, these tend to be few and far between so a couple of paragraphs is sufficient for this purpose.

Instead of mixing our `EFT` class into `CheckingAccount`, we could have made it a superclass and used multiple inheritance instead. Or even modified or derived from the `CheckingAccount` class to add transfer methods. Why did we choose to go the mix-in route?

Not directly inheriting or modifying the `CheckingAccount` class was a no-brainer for obvious reasons. The functionality is something that could be used for other account types as well and it does not make sense to duplicate code and add it to every class that needs those features. That leaves the question of multiple inheritance.

There were several considerations:

- Inheritance implies an *is-a* relationship between classes. Saying a checking account is-a "account that has transfer features" sounds somewhat contrived.
- The above stems from the fact that EFT does not really reflect a real object. It is more like a set of features or capabilities that accounts have. In the real world, it would be a checkbox on an account opening form for a checking account. The general thinking is that such classes are better modeled as mix-ins.
- Perhaps most important, when implemented as a mix-in, we can provide the feature sets to individual accounts, for example to specific savings accounts. You cannot use multiple inheritance to specialize individual objects in this manner.

For these reasons, mix-ins seemed a better choice in our design (aside from the fact that we needed **some** example to illustrate mix-ins).

There is one practical aspect of TclOO design that may drive your decision. Methods implemented via mix-ins appear in the method chain (Section 18.9) **before** methods defined on the object whereas inherited methods appear **after**. This is only relevant if the mix-in overrides existing methods.

18.7. Method forwarding

```
forward METHOD TARGET ?ARG …?
```

A method can be *forwarded* to another command, its *target*, so that when the method is invoked on the object, the target command is invoked instead. Forwarded methods may be defined on a class or on an object using the `forward` command.

METHOD is the name of the method to be defined. *TARGET* is the command to be invoked when that method is invoked. There are no restrictions on what *TARGET* may be. It may be a Tcl procedure or command, another object, a coroutine etc. It is invoked within the context of the object so that name resolution occurs in that context. So for example, if *TARGET* is `my`, the method is effectively forwarded to another method defined for the same object.

When *METHOD* is invoked, any optional *ARG* arguments specified in the `forward` declaration are prepended to the list of arguments provided by the caller before being passed to the target command.

Method forwarding is used in many patterns in object-oriented programming such as composition. Earlier we defined the cash management account using multiple inheritance, in effect treating a `CashManagementAccount` as being a `CheckingAccount` and a `BrokerageAccount`. We could instead have thought of it as a consolidated account that **contained** the two account types. The class would then be defined as

```
oo::class create ConsolidatedAccount {
    constructor {acct_no} {
        CheckingAccount create checking_account $acct_no
        BrokerageAccount create brokerage_account $acct_no
    }
}
```

We want the same operations available for this new account type as before. We can do this by forwarding methods to the appropriate contained account. For example, cash withdrawals would happen from the checking account.

```
oo::define ConsolidatedAccount {
    forward buy brokerage_account buy
    forward sell brokerage_account sell
    forward cash_check checking_account cash_check
    forward withdraw checking_account withdraw ❶
}
```

❶ Note we want withdrawals to be from the checking account

When the methods are forwarded, the target commands are resolved within the object's context. Thus `brokerage_account` and `checking_account` refer to the account objects we created within the object's context in the constructor.

This now behaves like our `CashManagementAccount` based solution.

```
% ConsolidatedAccount create consolidated CONS-0000001
→ Reading account data for CONS-0000001 from database
  Reading account data for CONS-0000001 from database
  ::consolidated
% consolidated cash_check Payee 500
→ Writing a check to Payee for 500
% consolidated buy GOOG 100
→ Buying 100 shares of GOOG
```

Forwarding may also supply arguments to the targeted command as shown below.

```
oo::objdefine consolidated {
    forward quick_cash my withdraw 100
}
consolidated quick_cash
→ 9400
```

For illustrative purposes, this last definition differs from previous ones in the following respects:

- The forwarded method is only defined for the object, not the class.
- It is forwarded to another method in the same object through the use of the `my` command.
- It supplies an argument within the forwarding definition.

18.8. Filter methods

```
filter ?SLOTOPERATION? ?METHOD …?
```

Imagine Mr. Smith is suspected of being up to his old tricks again and we need to monitor his accounts and log all activity. How would we do this? We could specialize every method for his accounts via `oo::objdefine` and log the activity before invoking the original method. We would have to do this for every method available to the object — those defined in the object, its class (and superclasses), object mix-ins and class mix-ins. This would be tedious and error prone. Moreover, since Tcl is a dynamic language, we would have to make sure we do that any time new methods were defined for the object or any ancestor or mix-in.

Filter methods offer a easier solution. A filter method is defined in the same manner as any method in the class or object. It is marked as a filter method using the `filter` command. Any method invocation on the object will then result in the filter method being invoked first.

The command installs all listed methods as filters in the manner specified by *SLOTOPERATION* (Section 18.2.6). By default, the methods are appended to the existing list of filters.

We can add a filter method to the account object whose activity we want to track.

```
oo::objdefine smith_account {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
    filter Log
}
```

We use the `info level` command to show the stack level. The `self target` command (Section 18.12.11) returns the target method name and its defining class.

Now all actions on the account will be logged.

```
% smith_account deposit 100
→ Log(1): 2-71828182 ::Account deposit: 100
  Log(2): 2-71828182 ::Account UpdateBalance: 100
  9900
```

Notice from the output that all method invocations, even those called internally from `deposit` are recursively logged. The filter method must therefore be aware that it may be recursively entered.

Some additional notes on filter methods:

- When methods are chained, the filter is called for **every** method in the chain.
- Multiple filters may be present and are chained like any other method.
- Because filter methods are called for **all** method invocations, they are generally defined with a variable number of arguments.
- Filter methods may be defined on an object, as in our example, or on a class, in which case they will affect all objects belonging to the class.
- Our filter method is not exported because it starts with an upper-case letter. This means it will not be called accidentally by clients of the object. However, there is no requirement that filter methods must be unexported.
- The filter method normally passes on the call to the target method via `next` which can be called at any point in the filter method. Moreover, the filter is not required to call the target method at all.
- The filter method may return as its result the target method result, or something else entirely.
- Filter methods are bypassed when invoking constructors, destructors or the `unknown` method of a class.

18.8.1. Defining a filter class

The `filter` declaration need not occur in the same class that defines the filter method. This means you can define a generic class for a filter which can be mixed into a “client” class or object which can install or remove the filter at appropriate times as desired.

Let us rework our previous example. To start with a clean slate, let us get rid of the `Log` method defined earlier.

```
oo::objdefine smith_account {
    filter -clear ❶
    deletemethod Log
}
```

❶ Note `-clear` option to clear any currently defined filters

Then we define a class `Logger` that does the logging. Since we only want transactions for Mr. Smith’s account to be logged, we mix it into the object and add the filter.

```
oo::class create Logger {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
}
oo::objdefine smith_account {
    mixin Logger
    filter Log
}
smith_account withdraw 500
→ 9400
  Log(1): 2-71828182 ::Account withdraw: 500
  Log(2): 2-71828182 ::Account CheckMinimum: 500
  Log(2): 2-71828182 ::Account UpdateBalance: -500
```

As you can see, we have the same behaviour as before. The advantage is that defining a class allows a collection of additional behaviours to be abstracted and easily added to any class or object without repeating the code.

18.8.2. When to use filters

Filters could be replaced by other techniques such as overriding and then chaining methods. Conversely, method overrides, such as in our account `freeze` example, could be replaced by filters. Usually though, it is clear which one makes the most sense.

Some general rules are:

- If we need to hook into multiple methods, it is easier to use a filter method rather than overriding individual methods. If necessary, `self target` can be used within the filter to selectively hook specific methods as illustrated in Section 18.12.11.
- When a method behaves more as an "observer" on an object as opposed to being a core part of the object's function, a filter method is a better fit.
- Filter methods are always placed at the front of the method chain so that can be a factor as well in deciding to use a filter.

18.9. Method chains

Throughout this chapter we have seen that when a method is invoked on an object, the code implementing the method for that object may come from several different places — the object, its class or an ancestor, a mix-in, forwarded methods, filters or even `unknown` method handlers. TclOO locates the code to be run by searching the potential implementations in a specific order called the *method chain*. It then runs the first implementation in this list. That implementation **may** choose to invoke the next implementation in the method chain using the `next` or `nextto` commands (Section 18.4.1.1) and so on through the list.

18.9.1. Method chain order

```
info object call OBJECT METHOD
```

For the exact search order and construction of a method chain, see the reference documentation of the `next` command. Here we will simply illustrate with an example where we define a class hierarchy with multiple inheritance, mix-ins, filters and object-specific methods.

The method definitions in our example are empty because we are not actually going to call them. The `info object call` command will give us the information we seek. It returns a list containing the method chain for the *`METHOD`* in object *`OBJECT`*. Each element of the list is a sublist with four items:

- the type, which may be `method` for normal methods, `filter` for filter methods, `private` for private methods or `unknown` if the method was invoked through the `unknown` (Section 18.2.5.4) facility
- the name of the method which, as noted from the output, may not be the same as the name used in the invocation
- the source of the method, for example, a class name where the method is defined, or the literal string `object` for methods defined on the instance
- the *implementation* type of the method as returned by the `info object methodtype` command (Section 18.12.8)

```
oo::class create ClassMixin { method m {} {} }
oo::class create ObjectMixin { method m {} {} }
oo::class create Base {
    mixin ClassMixin
    method m {} {}
    method classfilter {} {}
    filter classfilter
    method unknown args {}
}
oo::class create SecondBase { method m {} {} }
oo::class create Derived {
    superclass Base SecondBase
    method m {} {}
}
```

Having defined our classes, let us create an object and add in some object-specific methods and mix-ins.

```
Derived create o
oo::objdefine o {
    mixin ObjectMixin
    method m {} {}
    method objectfilter {} {}
    filter objectfilter
}
```

We have created an object of class `Derived` that inherits from two parent classes, all of which define a method `m`. Further we have mix-ins for both the class and the object. To confuse matters further, we have filters defined at both the class and object levels.

What will the method chain for method `m` look like? Luckily, we do not have to work it out while reading the manpage. We can do it through introspection using `info object call`.

```
% print_list [info object call o m]
→ filter objectfilter object method
  filter classfilter ::Base method
  method m ::ObjectMixin method
  method m ::ClassMixin method
  method m object method
  method m ::Derived method
  method m ::Base method
  method m ::SecondBase method
```

Study the output carefully to note the order. We can see for example that the filter methods appear at the head of the list, mix-ins are prioritized before objects and classes, and definitions in objects are prioritized over those in classes.

We reiterate that not every method in the chain is automatically invoked. Whether a method occuring in the list is actually called or not will depend on preceding methods passing on the invocation via the `next` or `nextto` commands.

18.9.2. Method chain for unknown methods

The method chain for undefined methods can be retrieved the same way.

```
% print_list [info object call o nosuchmethod]
→ filter objectfilter object method
  filter classfilter ::Base method
  unknown unknown ::Base method
  unknown unknown ::oo::object {core method: "unknown"}
```

As expected, the `unknown` method (Section 18.2.5.4), where defined, is called. Note the ancestor of **all** TclOO objects, `oo::object`, has a predefined `unknown` method.

18.9.3. Retrieving the method chain for a class

```
info class call CLASS METHOD
```

The `info class call` command returns the method chain for methods defined in a class.

```
% print_list [info class call Derived m]
→ filter classfilter ::Base method
  method m ::ClassMixin method
  method m ::Derived method
  method m ::Base method
  method m ::SecondBase method
```

18.9.4. Inspecting method chains within method contexts

```
self call
```

Within a method context, the command `self call` returns more or less the same information for the current object as `info object call`.

In addition, you can use `self call` from within a method context to locate the current method in the method chain. This command returns a pair, the first element of which is the same as the method chain list as returned by `info class call` command. The second element is the index of the current method in that list.

An example will make this clearer.

```
% oo::class create Base {
    constructor {} {puts [self call]}
    method m {} {puts [self call]}
}
→ ::Base
% oo::class create Derived {
    superclass Base
    constructor {} {puts [self call]; next}
    method m {} {
        puts [self call]; next
    }
}
→ ::Derived
% Derived create o
→ {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 0
  {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 1
  ::o
% o m
→ {{method m ::Derived method} {method m ::Base method}} 0
  {{method m ::Derived method} {method m ::Base method}} 1
```

Note the special form `<constructor>` for constructors. Although not shown in our example, destructors similarly have the form `<destructor>`.

Constructor and destructor method chains are only available through `self call`, not through the `info class call` command.

18.9.5. Looking up the next method in a chain

```
self next
```

At times a method implementation may wish to know if it is the last method in a method chain and if not, what method implementation will be invoked next. This information can be obtained with the `self next` command from within a method context. We illustrate by modifying the `m` method of the `Derived` class that we just defined.

```
% oo::define Derived {
    method m {} { puts "Next method in chain is [self next]" }
}
% o m
→ Next method in chain is ::Base m
```

As seen, `self next` returns a pair containing the class or object implementing the next method in the method chain and the name of the method (which may be `<constructor>` and `<destructor>`). In the case the current method is the last in the chain, an empty list is returned.

Notice above that although the next method in the method chain is printed out, it does not actually get invoked because the `m` method in `Derived` no longer calls `next`.

Do not confuse `self next` with `next`. The former only returns the next method is while the latter actually invokes it.

There is one important issue solved by `self next` that we will illustrate with an example. Imagine we want to package some functionality as a mix-in class. The actual functionality is immaterial but it is intended to be fairly general purpose (for example, logging or tracing) and mixable into any class.

```
oo::class create GeneralPurposeMixin {
    constructor args {
        puts "Initializing GeneralPurposeMixin";
        next {*}$args
    }
}
oo::class create MixerA {
    mixin GeneralPurposeMixin
    constructor {} {puts "Initializing MixerA"}
}
MixerA create mixa
→ ::mixa
  Initializing GeneralPurposeMixin
  Initializing MixerA
```

So far so good. Now let us define another class that also uses the mix-in.

```
% oo::class create MixerB {mixin GeneralPurposeMixin}
→ ::MixerB
% MixerB create mixb
Ø Initializing GeneralPurposeMixin
  no next constructor implementation
```

Oops. What happened? If it is not clear from the error message, the issue is that the GeneralPurposeMixin class naturally calls `next` so that the class that mixes it in can get initialized through its constructor. The error is raised because class `MixerB` does not have constructor so there is no “next” method (constructor) to call.

This is where `self next` can help. Let us redefine the constructor for GeneralPurposeMixin.

```
oo::define GeneralPurposeMixin {
    constructor args {
        puts "Initialize GeneralPurposeMixin";
        if {[llength [self next]]} {
            next {*}$args
        }
    }
}
MixerB create mixb
→ ::mixb
  Initialize GeneralPurposeMixin
```

It all works now because we only call `next` if there is in fact a next method to call.

18.9.6. Controlling invocation order of methods

```
nextto CLASSNAME ?args?
```

As we have seen in our examples, a method can use the `next` command to invoke its successor in the method chain. With multiple inheritance, mix-ins, filters involved, it may sometimes be necessary to control the order in which inherited methods are called. The `next` command, which goes strictly by the order in the method chain, is not suitable in this case.

The `nextto` command allows this control. It is similar to `next` except that it takes an argument that specifies the name of the class that implements the next method to be called.

Here *`CLASSNAME`* must be the name of a class that implements a method appearing later in the method chain.

When might you use this ? Well, imagine you define a class that inherits from two classes whose constructors take different arguments. How do you call the base constructors from the derived class? Using `next` would not work because the parent class constructors do not take the same arguments. We can use `nextto` instead as illustrated below.

```
oo::class create ClassWithOneArg {
    constructor {onearg} {puts "Constructing [self class] with $onearg"}
}
oo::class create ClassWithNoArgs {
    constructor {} {puts "Constructing [self class]"}
}
oo::class create DemoNextto {
    superclass ClassWithNoArgs ClassWithOneArg
    constructor {onearg} {
        nextto ClassWithOneArg $onearg
        nextto ClassWithNoArgs
        puts "[self class] successfully constructed"
    }
}
```

We can now call it without conflicts.

```
% DemoNextto create demo "a single argument"
→ Constructing ::ClassWithOneArg with a single argument
  Constructing ::ClassWithNoArgs
  ::DemoNextto successfully constructed
  ::demo
```

18.10. Programming without classes

Our exposition of object-oriented programming in Tcl has so far been centered around classes. Behaviours, object construction, inheritance and other relationships are all expressed in terms of classes.

However, not all object-oriented programming models involve classes. Another style, refered to in various forms as *classless* or *prototype-based programming*, dispenses with classes completely. Instead, objects are *cloned* from other objects, called prototypes, with the same methods and configuration. The inheritance features in class-based systems are replaced by the ability to add, remove or otherwise modify methods and delegate others.

As we stated in our introductory chapter, one of the defining features of TclOO is its flexibility in being adaptable to different programming models. Here we present classless object-based programming as one example of this.

We have already seen one of the requirements for such a system — the ability to define methods at an individual object level. We now describe how TclOO fulfils two additional ones — creating objects outside of classes, and cloning of objects.

The first of these is very straightforward, given our earlier discussion of `oo::object`. Given its dual nature as a class as well as an object, we can use it for creating a classless object.

```
oo::object create oa → ::oa
```

Strictly speaking, this object does have a class as seen below.

```
info object class oa → ::oo::object
```

For practical purposes though we can still treat this as classless as we did not have to go through an explicit class definition.

We now have what is essentially a shell of an object. The only method available for the object is `destroy` which it inherits from `oo::object`. There are no object-specific methods defined on it yet.

```
info object methods oa → (empty)
```

We now need to fill the object with methods and data. We know how to do that with `oo::objdefine`.

```
oo::objdefine oa {
    variable x
    method setx {val} {set x $val}
    method getx {} {set x}
}
oa setx 100
→ 100
```

We now have a functioning object. The last requirement is the ability to clone the object. The `oo::copy` command does this for us.

```
oo::copy OBJ ?NEWOBJ?
```

The command creates a new object named *`NEWOBJ`* that is a copy of *`OBJ`*. If *`NEWOBJ`* is not specified, the new object is created with an automatically generated name. The new object is of the same class as the source object, includes any object-specific methods that were defined on it, and contains the same variables with values as of the time of copying.

```
oo::copy oa ob → ::ob
ob getx        → 100
```

We are now free to extend (or not) this new object as we wish.

```
oo::objdefine ob method doublex {} {incr x $x} → (empty)
ob doublex                                     → 200
```

Notice that we now have similar functionality to method inheritance and overriding in class-based systems. A full-blown prototype-based object system would need more machinery than described here but that is all implementable with the mechanisms we have described so far and the introspection capabilities we describe in Section 18.12.

One final point about `oo::copy` that is worth noting. In some cases, copying across methods and variable definitions may not suffice for cloning an object. For example, the source object may have a variable that is being traced or a file that is open. To deal with such cases, the object may define a `<cloned>` method. This will be invoked at the time the object is cloned and is passed the source object as its sole argument. This method can then do any additional work required to create a full clone.

18.11. Metaclasses

TclOO supports the concept of *metaclasses*. Just as classes encapsulate the behaviour of class instances (objects), metaclasses do the same for classes. So in effect metaclasses are just classes whose instances are themselves classes.

The concept can be a little confusing so let us start by revisiting how classes are created. The session below uses some TclOO introspection commands that are covered in Section 18.12 but their meaning should be obvious from the context.

Consider the following:

```
oo::class create MyClass {method m {} {puts "m called"}} → ::MyClass
MyClass create myObject                                   → ::myObject
info cmdtype myObject                                     → object
info cmdtype MyClass                                      → object
```

The similarity between creation of the class `MyClass` and its instance `myObject` (the `destroy` method would also show the analogy) should hint at something that is confirmed by introspecting their command type (Section 3.5.7). We see that `MyClass` is itself an object, i.e. an instance of some class. But of what class? Further introspection reveals

```
info object class myObject → ::MyClass
info object class MyClass  → ::oo::class
```

So `MyClass` is an instance of the class `oo::class`. But going one level deeper, if `oo::class` is a class, it must itself be an instance of some class as we saw above. So what class is `oo::class` an instance of?

```
info cmdtype oo::class      → object
info object class oo::class → ::oo::class
```

The buck stops here; `oo::class` is an instance of itself! We can look at the methods that can be invoked on it.

```
info class methods oo::class -all  → create destroy new ❶
info object methods oo::class -all → create destroy ❷
```

❶ Methods that be invoked on **instances** of `oo::class`, for example `MyClass`
❷ Methods that be invoked on `oo::class` **itself**

This is all a little head spinning so let us summarize and examine the implications in the context of our metaclass discussion.

- Classes are themselves objects that are instances of the `oo::class` class.
- `oo::class` is a class and because its instances are themselves classes, `oo::class` is also a *metaclass*, a class that creates classes.

This last point raises the following question. Given that `oo::class` can create instances that are metaclasses (the proof being `oo::class` is a metaclass and an instance of itself), can it create other metaclasses as well? The answer is yes, as we describe in the next section.

First though, what is the motivation behind the desire for additional metaclasses and why is `oo::class` not sufficient? The answer is that `oo::class` is in fact sufficient but the additional metaclasses permit the following features without having to repeat or duplicate code.

- Additional restrictions may be placed on creation and deletion of classes and objects (Section 18.11.2, Section 18.11.3).
- Additional or restricted commands can be used within a class definition script (Section 18.11.1, Section 18.11.4).

Tcl itself provides other metaclasses in addition to `oo::class`. These are covered in later sections but first we will look at how to write our own.

18.11.1. Implementing a metaclass

Suppose we wanted our definition language to automatically create **getter** and **setter** methods that would allow clients of the class to retrieve values of specific variables without having to write individual methods. A sample class definition might look like

```
CustomMetaclass create Employee {
    properties name salary
    method raise {increment} {incr salary $increment}
}
```

The `properties` command within the `Employee` class definition should suffice to create variables `name` and `salary` with methods `get_name`, `set_name` etc. that retrieve or set the values of the corresponding variables.

We cannot use `oo::class` to define such a class because (obviously) it has no idea what to do with the `properties` command in the class definition script. We will therefore have to write a metaclass, `CustomMetaclass`, that extends `oo::class`.

We first define a namespace `property_impl` that will be used as the context for the definition script for `CustomMetaclass`. This namespace will contain the implementation of the `properties` command. Since the standard commands used in definition scripts, like `method` and `forward` need to also be available, the namespace path in `property_impl` is set up to include `oo::define`.

```
namespace eval property_impl {
    namespace path ::oo::define
    proc properties {args} {
        uplevel 1 [list variable {*}$args]
        foreach arg $args {
            uplevel 1 [list method get_$arg {} "return \[set $arg\]"]
            uplevel 1 [list method set_$arg {val} "set $arg \$val"]
        }
    }
}
```

The `properties` command creates the method definitions for the getters and setters. The `uplevel` is required so the `method` command is invoked in the context of the class definition script (the last argument to the create method).

The second step is to define our metaclass which understands the use of `properties` in class definition scripts.

- This metaclass needs to support all other capabilities of `oo::class` and therefore inherits from it.
- By default, class definition scripts execute in the `oo::define` namespace context. That needs to be changed to our `property_impl` namespace so the `properties` command comes into scope. The `definitionnamespace` command has the desired effect.

```
oo::class create CustomMetaclass {
    superclass oo::class
    definitionnamespace ::property_impl
}
```

We can now define classes whose definition scripts can include `properties` in addition to the standard class definition commands.

```
CustomMetaclass create Employee {
    properties name salary

    method raise {increment} {incr salary $increment}
}
```

And to try it out,

```
Employee create joe  → ::joe
joe set_name Joe     → Joe
joe set_salary 35000 → 35000
joe raise 5000       → 40000
joe get_salary       → 40000
```

Of course, the metaclass is general purpose and usable for other classes.

```
CustomMetaclass create City {
    properties Name Population
    constructor {name population} {
        set Name $name
        set Population $population
    }
}
City create lagos Lagos 17000000
lagos get_Population
→ 17000000
```

18.11.2. Abstract classes: oo::abstract

Having looked at how a metaclass can be implemented, we will take a look at the metaclasses that are already available in Tcl starting with `oo::abstract`. This class implements the common concept of abstract classes in OO programming — classes intended to be used as strictly as base classes and whose instances cannot be directly created. Abstract classes provide the base methods common to subclasses while expecting derived classes to implement specialized ones.

As an example, assume our application deals with regular polygons and must know the perimeter and area of each. The `RegularPolygon` abstract class provides the method `perimeter` as we can calculate that knowing the number of sides and the length of each. However, not knowing about apothems, we do not know the general formula for area which

must be therefore implemented by subclasses that model polygons of a specific number of sides.

```
oo::abstract create RegularPolygon {
    variable Sides Len
    constructor {sides len} {
        set Sides $sides
        set Len $len
    }
    method perimeter {} {
        return [expr {$Sides * $Len}]
    }
}
oo::class create Square {
    superclass RegularPolygon
    constructor {len} {
        next 4 $len
    }
    method area {} {
        my variable Len
        return [expr {$Len * $Len}]
    }
}
```

An attempt to create an instance of the abstract class `RegularPolygon` fails.

```
% RegularPolygon create poly
Ø unknown method "create": must be destroy
```

However, we can create objects of the derived class and call inherited methods on it as usual.

```
% Square create square 5
→ ::square
% square area
→ 25
% square perimeter
→ 20
```

Abstract classes in Tcl cannot be used to enforce interface requirements on child classes. So there is no way to say for example that every derived class must have an `area` method.

18.11.3. Singleton classes: oo::singleton

The singleton pattern is employed when you want at most one instance of a class to exist to ensure consistency among all components in an application. Use cases may include a common message logging facility, an application settings store, an access control database and others.

In TclOO, the `oo::singleton` metaclass creates singleton classes. Here is simple singleton for checking access.

```
oo::singleton create AccessControl {
    constructor {} {
        link {::check_access check} ❶
    }
    method check {user} { return [string equal $user "goodguy"] }
}
```

❶ For convenience, not mandated. See text.

To create an instance of a singleton, its `new` method must be called. As seen below, this always returns the same object. The `create` method is not exported and the `destroy` method or `oo::copy` cannot be called on the instance.

```
% set accessControl [AccessControl new]
→ ::oo::Obj1055
% AccessControl new
→ ::oo::Obj1055
% AccessControl new
→ ::oo::Obj1055
% AccessControl create
Ø unknown method "create": must be destroy or new
% $accessControl destroy
Ø may not destroy a singleton object
```

While the `destroy` method cannot be called, the instance can still be destroyed by other means such as renaming it to the empty string, deleting the containing namespace or destroying the class itself.

Callers need not know or store the name of the access control object, and can instead directly invoke `check` on the object returned by `new`. For a bit of convenience, the constructor used the `link` command to expose an equivalent global command.

```
[AccessControl new] check goodguy → 1
check_access badguy               → 0
```

There are some caveats to keep in mind with singletons:

- Their constructors should not accept any arguments as multiple calls to `new` with differing arguments may lead to inconsistency.
- Classes inheriting from a singleton class are not themselves singletons. This is generally not recommended.

18.11.4. Configurable properties: oo::configurable

The last built-in metaclass we describe, `oo::configurable` serves a very similar purpose to the custom metaclass we implemented in Section 18.11.1. It introduces a `property` command that can be used within a class definition script. These properties can be then accessed by

client code using a `configure` method in the same manner as the `chan configure` command for channels.

Let us recreate our Employee class using `oo::configurable`.

```
oo::configurable create Employee {
    property name salary -kind readable nickname
    constructor {args} {
        my configure {*}$args
        my variable salary ❶
        set salary 35000
    }
    method raise {increment} {
        my variable salary
        incr salary $increment
    }
}
Employee create joe -name Joseph -nickname Joseph
→ ::joe
```

❶ Property has to be explicitly brought into scope with `variable` in the class definition script or `my variable` in the method body. Alternatively, retrieve its value with `my configure`.

The `property` command has the syntax

```
property ?PROPERTYNAME ?OPTION …? ? …
```

Multiple properties may be listed, with each followed by options applicable to it, if any. The property names must not begin with a `-` character.

Supported options are shown in Table 18.4.

Table 18.4. oo::configurable property command options

Option	Description
`-get` *`SCRIPT`*	Defines *`SCRIPT`* as the body of the method used to retrieve the property value. Defaults to getting the value of a variable of the same name as the property.
`-kind` *`PERM`*	*`PERM`* should be one of `readable`, `writable` or `readwrite` (default).
`-set` *`SCRIPT`*	Defines *`SCRIPT`* as the body of the method used to set the property value. Defaults to setting the value of a variable of the same name as the property.

The options above are not discussed further here as their semantics and usage should be obvious. See the `oo::configurable` reference documentation for details.

The `configure` method can retrieve all properties, a single property or to set the values of properties. The property names need to be prefixed with a `-` when used as options to `configure`.

```
joe configure                → -name Joseph -nickname Joseph -salary 35000
joe configure -nickname Joe → (empty)
joe configure -salary       → 35000
joe configure -salary 40000 Ø property "-salary" is read only ❶
joe raise 5000              → 40000 ❷
joe configure               → -name Joseph -nickname Joe -salary 40000
```

❶ Property is read-only ...
❷ ...but can be changed within a method

The `oo::configuresupport` namespace, not covered in this book, contains several support commands internally used in the implementation of the `oo::configurable` class. You can used these to implement your own custom metaclasses with similar characteristics to `oo::configurable`.

18.12. OO introspection

Introspection of classes and objects from any context is primarily accomplished through the `info class` and `info object` ensemble commands. In addition, the `self` command can be used for introspection of an object from inside a method context for that object.

18.12.1. Enumerating objects and classes

```
info class instances CLASS ?PATTERN?
```

The `info class instances` command returns the objects belonging to the specified class.

```
info class instances Account        → ::oo::Obj966 ::smith_account
info class instances SavingsAccount → ::savings
```

As seen above, this command will only return objects that directly belong to the specified class, not if the class membership is inherited.

You can optionally specify a pattern argument in which case only objects whose names match the pattern using the rules of the `string match` command are returned. This can be useful when namespaces are used to segregate objects.

```
info class instances Account ::oo::* → ::oo::Obj966
```

Classes are also objects in TclOO and therefore `info class instances` can be used to enumerate classes.

```
% info class instances oo::class
→ ::oo::object ::oo::class ::oo::Slot ::oo::configuresupport::configurable ::oo...
```

We pass `oo::class` to the command because that is the class that all classes are instances of. The returned list contains two interesting elements:

- `oo::class` is returned because as we said it is a class itself (the class that all class objects belong to) and is therefore an instance of itself.
- If that were not confusing enough, `oo::object` is also returned. This is the root class of the object hierarchy and hence is an ancestor of `oo::class`. At the same time it is a class and hence must be an instance of `oo::class` as well.

This circular and self-referential relationship between `oo::object` and `oo::class` seems strange but it is what allows all programming constructs in TclOO to be work in consistent fashion. It is also a common characteristic of many OO systems.

18.12.2. Checking if an object is a class

```
info object isa class OBJECT
info object isa metaclass OBJECT
```

The `info object isa class` command returns `1` if *`OBJECT`* is a class and `0` otherwise. A class is an instance of `oo::class` or one of its subclasses.

```
info object isa class SavingsAccount → 1
info object isa class savings        → 0 ❶
info object isa class clock          → 0 ❷
```

❶ An object but not a class
❷ A command and not a class

A related command is `info object isa metaclass` which returns `1` if the passed argument is a class that can create classes.

```
info object isa metaclass oo::class       → 1
info object isa metaclass CustomMetaclass → 1
info object isa metaclass Account         → 0
```

18.12.3. Inspecting class relationships

```
info class mixins CLASS
info class subclasses CLASS PATTERN
info class superclasses CLASS
```

The `info class superclasses` command returns the **direct** superclasses of a class in order of inheritance precedence. Notice below that `oo::object` is a superclass of `oo::class`.

```
% info class superclasses CashManagementAccount
→ ::CheckingAccount ::BrokerageAccount
% info class superclasses ::oo::class
→ ::oo::object
```

Conversely, the `info class subclasses` will return the classes directly inheriting from the specified class. If *`PATTERN`* is specified, only subclasses whose names match *`PATTERN`* using `string match` rules are returned.

```
% info class subclasses Account
→ ::CheckingAccount ::SavingsAccount ::BrokerageAccount
% info class subclasses Account *k*
→ ::CheckingAccount ::BrokerageAccount
```

As one might expect, there is also a command, `info class mixins` for listing mix-ins.

```
% info class mixins CheckingAccount
→ ::EFT
```

18.12.4. Retrieving class definition namespaces

```
info class definitionnamespace CLASS ?KIND?
```

The `info class definitionnamespace` returns the definition namespace (Section 18.11.1) for *`CLASS`*. The namespace returned corresponds to either `oo::define` or `oo::objdefine` depending on whether *`KIND`* is `-class` (default) or `-instance`. The command returns an empty string if the class does not provide a definition space of the specified kind.

```
info class definitionnamespace CustomMetaclass           → ::property_impl
info class definitionnamespace CustomMetaclass -class    → ::property_impl
info class definitionnamespace CustomMetaclass -instance → (empty)
```

18.12.5. Object identity

```
info object creationid
info object namespace
self namespace
self object
```

Under some circumstances, an object needs to discover its own identity from within its own method context, for example

- when an object method has to be passed to a command callback
- when an object is redefined “on the fly” from within an method, its name must be passed to `oo::objdefine`. See Section 18.5.2 for an example.

The `self object` command, which can also be called simply as `self`, returns this information. We have seen this used in several times in this chapter.

In addition to the command used to access it, an object may also be identified by the unique namespace in which the object state is stored. This is obtained through the `self namespace`

command within a method context or with the `info object namespace` command from outside that context.

```
% oo::define Account {method get_ns {} {return [self namespace]}}
% savings get_ns
→ ::oo::Obj977
% set acct [Account new 0-0000000]
→ Reading account data for 0-0000000 from database
  ::oo::Obj1059
% $acct get_ns
→ ::oo::Obj1059
% info object namespace $acct
→ ::oo::Obj1059
```

Notice that when we create an object using `new`, the namespace matches the object command name. This is an artifact of the implementation and this should not be relied on. In fact, like any other Tcl command, the object command can be renamed.

```
% rename $acct temp_account
% temp_account get_ns
→ ::oo::Obj1059
```

As you can see, the object command and its namespace name no longer match. Also note that the namespace does not change when the object is renamed.

Every object also has a creation id that is unique within an interpreter and never changed. It can be retrieved with `info object creationid`.

```
info object creationid temp_account → 1059
```

While the creation id may be part of the object namespace in some Tcl versions, there is no guarantee it will remain so in the future.

18.12.6. Inspecting an object's class membership

```
info object class OBJECT ?CLASS?
info object isa typeof OBJECT CLASS
info object isa mixin OBJECT CLASS
info object mixins OBJECT
self class
```

If the *`CLASS`* argument is not specified, `info object class` returns the class to which an object belongs.

```
info object class savings → ::SavingsAccount
```

If *`CLASS`* is specified, the command returns a boolean indicating whether the object belongs to the class taking inheritance into account.

```
info object class savings SavingsAccount  → 1
info object class savings Account         → 1
info object class savings CheckingAccount → 0
```

The `info object isa typeof` command is an alternate means of checking class membership.

```
info object isa typeof savings Account → 1
```

The `info object mixins` command, analogous to `info class mixins`, enumerates the classes mixed-in with an object.

```
info object mixins savings → ::EFT
```

Conversely, `info object isa mixin` checks if a class is **directly** mixed into an object.

```
info object isa mixin savings EFT → 1
```

From within the method context of an object, the `self class` command returns the class defining the currently executing method. **Note this is not the same as the class the object belongs to as the example below shows**.

```
oo::class create Base {
    method m {} {
        puts "Object class: [info object class [self object]]"
        puts "Method class: [self class]"
    }
}
oo::class create Derived { superclass Base }
Derived create o
o m
→ Object class: ::Derived
  Method class: ::Base
```

The `self class` command will fail when called from a method defined directly on an object since there will be no class associated with the method.

18.12.7. Checking if a command is an object

```
info object isa object CMD
```

The `info object isa object` command returns `1` if a command is an object and `0` otherwise.

```
info object isa object savings       → 1
info object isa object clock         → 0
info object isa object nosuchcommand → 0
```

18.12.8. Enumerating methods

```
info object methods CLASS ?-all? ?-private? ?-scope SCOPE?
info object methods OBJECT ?-all? ?-private? ?-scope SCOPE?
info class methodtype CLASS METHOD
info object methodtype OBJECT METHOD
```

The `info class methods` and `info object methods` commands retrieve the list of methods implemented by a class or object respectively. The commands return a list of methods **and** forwards exported by the class or object. By default, those defined in superclasses, mixins and from the object's class in the case of `info object`, are not included in this list.

```
info class methods CheckingAccount     → cash_check
info class methods ConsolidatedAccount → cash_check sell buy withdraw
info object methods consolidated       → quick_cash
```

You can distinguish between methods and forwards in the returned list by querying its type with the `info class methodtype` and `info object methodtype` commands. The commands will return `method` for the former and `forward` for the latter.

```
info class methodtype Account withdraw             → method
info class methodtype ConsolidatedAccount withdraw → forward
info object methodtype consolidated quick_cash     → forward
```

Passing the `-all` option will return inherited and mixed-in methods as well, and in the case of objects, methods defined in the object's class.

```
% info class methods CheckingAccount -all
→ balance cash_check deposit destroy getRoutingNumber get_ns setRoutingNumber t...
% info object methods consolidated -all
→ buy cash_check destroy quick_cash sell withdraw
```

If the `-private` option is passed, non-exported methods are also returned.

```
% info class methods CheckingAccount -private
→ cash_check
% info class methods CheckingAccount -all -private
→ <cloned> UpdateBalance balance cash_check deposit destroy eval getRoutingNumb...
```

The option `-private` refers to non-exported methods as opposed to the private methods (Section 18.4.4). This inconsistency is a historical artifact.

If the option `-scope` is specified, the command only returns methods defined directly within that class or object with the visibility passed as the option value. The option value may be `public`, `unexported` or `private` corresponding to exported, unexported and private methods respectively (Section 18.2.5.3). Note again, the semantics of `-scope private` differ from those of the `-private` option.

```
% info class methods Account -scope private
→ CheckMinimum
% info class methods Account -scope public
→ get_ns deposit setRoutingNumber getRoutingNumber balance withdraw
% info class methods Account -scope unexported
→ UpdateBalance
```

The `-scope` option is not available in Tcl 8.6.

18.12.9. Retrieving method definitions

```
info class constructor CLASS
info class definition CLASS METHOD
info class destructor CLASS
info class forward CLASS METHOD
info object definition OBJECT METHOD
info object forward CLASS METHOD
```

The `info class definition` command returns the definition of an non-forwarded method as a pair consisting of the method's arguments and its body. The method whose definition is being retrieved has to be defined in the specified class, not in an ancestor or a class that is mixed into the specified class.

```
% info class definition Account UpdateBalance
→ change {
          set Balance [tcl::mathop::+ $Balance $change]
          return $Balance
      }
```

Similarly, `info object definition` will return the definition of a method **directly** defined on an object. It will raise an error if passed a method name that is defined on the object's class.

Constructors and destructors are retrieved differently via `info class constructor` and `info class destructor`.

```
% info class constructor Account
→ account_no {
          puts "Reading account data for $account_no from database"
          set AccountNumber $account_no; # `AccountNumber` and `Balance` are in...
          set Balance 10000
      }
```

For methods that are forwards, the `info class forward` and `info object forward` return information about the forward definition in a class and object.

```
info class forward ConsolidatedAccount buy  → brokerage_account buy
info object forward consolidated quick_cash → my withdraw 100
```

18.12.10. Inspecting method chains and contexts

```
info class call CLASS METHOD
info object call OBJECT METHOD
self call
self caller
self method
self next
```

The `info class call` command retrieves the method chain for a method. From a method context, the `self call` command returns similar information while `self next` identifies the next method implementation in the chain. We have already discussed these in detail in Section 18.9. There are two additional related commands that provide further information within a method context.

- The `self method` command returns the name of the method being executed.
- The `self caller` command returns information of the caller of the method **when called from another method**. This is in the form of a list of three elements — the class, the object and the calling method.

```
oo::class create C {
    method m {} {
        lassign [self caller] cls obj meth
        puts "In [self method], called from method $meth in object $obj of class \
            $cls"
    }
    constructor {} { my m }
}
C create c
→ ::c
  In m, called from method <constructor> in object ::c of class ::C
```

18.12.11. Inspecting filters

```
info class filters CLASS
info object filters OBJECT
self filter
self target
```

The list of methods that are set as filters can similarly be obtained with `info class filters` or `info object filters`.

```
info object filters smith_account → Log
```

Note that `info object filters` will return a list of filter methods **directly** defined on the object. It will not include filters defined on the object's class.

When a method is run as a filter, it is often useful for it to know the real target method being invoked. This information is returned by `self target` which can only be used from within

a filter context. Its return value is a pair containing the declarer of the method and the target method name.

For example, suppose instead of logging every transaction as in our earlier example, we only wanted to log withdrawals. In that case we could have defined the `Log` command as follows:

```
oo::define Logger {
    method Log args {
        if {[lindex [self target] 1] eq "withdraw"} {
            my variable AccountNumber
            puts "Log([info level]): $AccountNumber [self target]: $args"
        }
        return [next {*}$args]
    }
}
```

We would now expect only withdrawals to be logged.

```
% smith_account deposit 100
→ 9500
% smith_account withdraw 100
→ Log(1): 2-71828182 ::Account withdraw: 100
  9400
```

The `self filter` command returns information about the filter itself. The command returns a list of three items:

- The name of the class or object where the filter is declared. Note this is **not** necessarily the same as the class in which the filter method is defined. Thus above, the filter was **defined** in the `Logger` class but **declared** in the `smith_account` object.
- Either `object` or `class` depending on whether the filter was declared inside an object or a class.
- The name of the filter itself.

```
% oo::define Logger {
    method Log args {
        puts [self filter]
        return [next {*}$args]
    }
}
% smith_account withdraw 1000
→ ::smith_account object Log
  ::smith_account object Log
  ::smith_account object Log
  8400
```

You will see multiple output lines in the above example. Remember the filter is called at *every* method invocation including nested calls.

18.12.12. Enumerating variables

```
info class variables CLASS ?-private?
info object variables OBJECT ?-private?
info object vars OBJECT ?PATTERN?
```

The command `info class variables` returns the list of variables that have been declared with the `variable` statement inside a class definition and are therefore automatically brought within the scope of the class's methods.

```
info class variables SavingsAccount → MaxPerMonthWithdrawals WithdrawalsThisMonth
```

The listed variables are only those defined through the `variable` statement **for that specified class**. Thus the above command will not show the variable `Balance` as that was defined in the base class `Account`, not in `SavingsAccount`.

For enumerating variables for an object as opposed to a class, there are two commands:

- `info object variables` behaves like `info class variables` but returns variables declared with `variable` inside object definitions created with `oo::objdefine`. These may not even exist yet if they have not been initialized.
- `info object vars` returns variables currently **existing** in the object's namespace and without any consideration as to how they were defined.

The example below illustrates the difference. The first command returns an empty list because no variables were declared through `variable` for **that object**. The second command returns the variables in the object's namespace. The fact that they were defined through a class-level declaration is irrelevant.

```
info object variables smith_account → (empty)
info object vars smith_account      → Balance AccountNumber
```

18.12.13. Enumerating configurable properties

```
info class properties CLASS ?-all? ?-readable? ?-writable?
info object properties OBJECT ?-all? ?-readable? ?-writable?
```

The `info class properties` and `info object properties` return the properties defined for a class or object (Section 18.11.4). By default, the commands return the names of readable properties defined directly in the class or object. If the `-all` option is specified properties of superclasses or mix-ins are also included. Specifying the `-writable` option returns writable properties **instead of** readable ones. A `-readable` option is also supported which has default behavior. If both options `-readable` and `-writable` are specified, the last takes effect.

```
info class properties Employee           → -name -nickname -salary
info class properties Employee -writable → -name -nickname
```

19

The Event Loop

> Great minds discuss ideas; average minds, events; small minds, people.
>
> — Eleanor Roosevelt

Unwilling to admit to a small mind, and not possessing of a great one, the author is left with no choice but to discuss events.

Most of the programming we have discussed so far involves sequential program flow where the Tcl interpreter executes a script a command at a time and terminates after the last command in the script. For applications like file search that are primarily command line utilities, this is adequate. For "long running" applications whose function is to react to different types of external events, this is not a suitable model. Common examples include GUI applications which react to events like mouse clicks and network services that respond to requests from multiple clients. These applications sit idle waiting for specific events to occur upon the occurence of which they execute *event handlers*. These event handlers may update the display in response to a key press, send back a web page to the requesting client and so on. These applications are said to be written in *event driven* style.

19.1. Event sources and types

Events may come from a variety of sources. The **built-in** sources in Tcl include

- Channels which generate events for I/O on the channel.
- Timers that generate events on the expiry of some interval.
- The idle loop which generates "pseudo-events" when the system has no other events to process. These are used by applications to schedule background tasks to be run when the system is otherwise idle.

Extensions may add their own event sources. The most commonly encountered ones are those added by the Tk graphical interface toolkit which sources events for display, mouse movement and keyboard input.

How you register event handlers is dependent on the event source generating the events. For example, `chan event` to register for channel events, `after` to schedule timer events etc.

In this chapter, we introduce Tcl's event loop, which is the framework for responding to events, and cover the `after` command used for scheduling timer and idle events. Event handling related to I/O is described in Chapter 21.

19.2. The Tcl event loop

Tcl provides built-in support for event-driven applications through its *event loop*. The basic flow of event driven applications in Tcl is as follows:

1. At startup, the application registers event handlers for events of interest.
2. The event loop is entered. This waits for events to occur and invokes the associated handler(s) when an event occurs.
3. Each handler takes the appropriate actions in response to the event. This could include registration of handlers for additional events or unregistration of existing handlers.
4. After the handler completes, control returns to the event loop in Step 2 and the cycle is repeated.

The second step above, entering the event loop, may be done by the application's native code outside of the Tcl interpreter, or from within a Tcl script through any of several commands like `update` (Section 19.2.3.2) or `vwait` (Section 19.2.3.1). Notice that the latter implies that event loops can be nested, something that we will look at in more detail later.

In case of a Tcl application running interpreters in multiple threads, each thread runs an independent event loop which is shared among all interpreters running in that thread.

19.2.1. The event and idle task queues

Internally, the event loop implementation maintains two queues:

- The *event queue*: on the occurence of a physical event, such as a mouse click or arrival of data on a channel, the driver or module responsible will place an entry in this queue with details of the event and the handler to be called for the event.
- The *idle task queue*: this second queue holds tasks to be executed when the system is idle, i.e. has no events to process. At the script level, tasks can be placed on this queue through the `after idle` command (Section 19.3.3).

19.2.2. Event loop operation

When the event loop runs,

1. It invokes the handlers for all entries present on the event queue. The handlers may themselves add new entries to the event queue and these are processed as well.
2. Once the event queue is empty, registered event sources are queried. These may place new events on the event queue in which case the event loop goes back to the first step.
3. If there are still no entries to process on the event queue, the event loop runs **all** the tasks present on the idle task queue.
4. If the idle task queue is empty, the event loop effectively blocks waiting for the next event to be placed on the event queue at which point it goes back to Step 1.

The above is a simplified and incomplete description adequate for our purposes.

19.2.3. Running the event loop

An application may itself enter the event loop from the C level. For example, the `wish` (Section 2.2.3) application evaluates the contents of any script file supplied as an argument and then enters the event loop processing user interface events. Thus in a `wish` based application, the event loop is effectively always running.

Alternatively, the Tcl script itself can initiate the event loop from script. This is commonly done from long running programs like network servers that are not GUI programs and therefore use `tclsh` (Section 2.2.2) rather than `wish` to host the interpreter.

Since we do not delve into C programming in this book, we will only concern ourselves with the latter method — running the event loop from a script.

19.2.3.1. Processing events based on conditions: vwait

```
vwait VARNAME
vwait ?OPTIONS? ?VARNAME …?
```

The `vwait` command runs the event loop, calling the handlers for each queued event in turn until specified conditions are satisfied at which point the command returns. If the event and idle queues are empty, the command blocks waiting for an event.

In the first form of the command, `vwait` will continue to process events until the variable *`VARNAME`* is written. *`VARNAME`* is resolved in the **global** context, and not in the context of the caller of `vwait`, so variables within a namespace must be fully qualified.

In event-driven Tcl applications such as network servers based on `tclsh`, a command similar to the following is commonly executed after initialization.

```
vwait forever
```

On execution of the above, `tclsh` will enter the event loop processing network connection requests until one of the event handlers either calls `exit` (Section 2.2.5) to terminate the application or sets the variable `forever` at which point the command returns. We will see an example of such a server in Chapter 21. Note that there is nothing special about the name `forever` above. It is commonly used as a hint that the command is supposed to loop forever and not return.

Here is a short script illustrating `vwait` that you can source into `tclsh`. The example uses the `after` command (Section 19.3) to schedule timer events that trigger after a specified interval.

```
after 1000 [list puts "1 second elapsed"]
after 2000 [list puts "2 seconds elapsed"]
after 3000 [list set ::done completed]
vwait ::done
puts $done
→ 1 second elapsed
  2 seconds elapsed
  completed
```

Running the above script, you will see the `tclsh` prompt disappear for three seconds. Because the event loop is running, the timer events are triggered at one second intervals. After 3 seconds the variable `done` is set causing the `vwait` command to stop its event loop and return.

The second syntactic form of `vwait` provides more flexibility.

This second form of `vwait` is not available in Tcl 8.6 and earlier.

In contrast to the first syntactic form which terminates execution of the event loop based on a single condition, writing to a variable, this second form allows the termination condition to be based on a combination of one or more conditions:

- readability or writability of zero or more I/O channels
- setting or unsetting of zero or more variables
- an optional timeout

The conditions on which to wait are specified through the options shown in Table 19.1. Multiple options may be passed and any option may be specified multiple times, to wait on multiple channels for example.

Table 19.1. Options controlling vwait conditions

Option	Condition
`-all`	Wait for all specified conditions to be satisfied. By default, the command will return when any condition is satisfied.
`-readable` *CHANNEL*	Channel `CHAN` is readable.
`-variable` *VARNAME*	The variable `VARNAME` is set or unset. This differs from the variables passed as additional arguments to the command in that the condition is also satisfied when the variable is unset.
`-timeout` *MILLISECS*	An interval of *MILLISECS* has elapsed. If multiple `-timeout` options are specified, the last one takes effect.
`-writable` *CHANNEL*	Channel `CHAN` is writable.

Another set of options, shown in Table 19.2, permits control of the types of events that will be processed allowing, for example, only file events to be processed.

Table 19.2. Options for vwait event types

Option	Description
`-nofilevents`	Skips I/O events.
`-noidleevents`	Skips events on the idle queue.
`-notimerevents`	Skips timer events.
`-nowindowevents`	Skips events from the windowing system

In the author's opinion, the options in Table 19.2 should be used with some care. Some I/O channels may also require idle or timer event processing.

The result of `vwait` is the empty string if neither `-timeout` nor `-extended` is passed. If option `-timeout` is passed, but not `-extended`, the return value the number of milliseconds remaining in the timeout when the command returns or `-1` if the timeout expired without any condition being met.

If the `-extended` option is specified, the return value is a list containing information about the met conditions. The elements at even indices in the list are one of `variable`, `readable`, `writable` or `timeleft` indicating the type of condition that was satisfied. The elements at the odd indices in the list are the variable name, channel name or remaining number of milliseconds depending on the condition. The order of elements is the same as the order in which the conditions were satisfied except for `timeleft` which always comes last.

19.2.3.1.1. Avoiding deadlocks with vwait

The `vwait` command should be used with some care from within code that runs within an event handler else deadlock can occur. Consider the following script.

```
proc handler {} {
    puts "handler enter"
    set ::varA 1
    vwait ::varB
    puts "handler exit"
}
proc demo {} {
    after 1000 handler
    vwait ::varA
    set ::varB 1
}
demo
```

If you run this fragment in a `tclsh` shell, you will find that the prompt does not return after printing `handler enter`. The expectation was that setting `varA` within `handler` would permit the `demo` procedure to continue which in turn would set `varB` allowing `handler` to also complete. However, what actually happens is that control will not return to `demo` until **after** `handler` completes. But `handler` cannot complete until `varB` is set. Hence deadlock.

This example is contrived but similar situations will arise in real life if you are not careful. The fundamental point to be made is that **nested calls to `vwait` do not execute "in parallel". The outermost call will not return until inner calls have returned. See the examples in the reference documentation of `vwait` for possible solutions.

These situations generally arise as a result of trying to make asynchronous execution appear synchronous. See the Tcl `vwait` reference documentation for additional information and workarounds. In many cases coroutines, which we discuss in Chapter 24, are a better option.

As an aside, the above example also illustrates that it is possible to recursively enter the event loop. You just have to take care not to deadlock.

19.2.3.2. Single invocation: update

```
update ?idletasks?
```

The `vwait` command (Section 19.2.3.1) command runs the event loop continuously until certain conditions are met and blocks, waiting for events, if necessary. In contrast, the `update` command invokes the event loop once and returns instead of blocking when there are no events to be processed.

If `idletasks` is not specified, the command enters the event loop and runs handlers for all entries on the event and idle task queues including new ones that may be added while the event loop is running. The command returns when both queues are empty. The following short example illustrates its behaviour.

```
proc handler {} {
    puts "Event 0"
    after 0 [list puts "Event 1"]
}
after 0 handler
after 1000 [list puts "Event 2"]
update
→ Event 0
  Event 1
```

The `after` command schedules a timer to expire after a specified time. Passing an argument value of `0` means the timer expires immediately while `1000` indicates expiry after one second. When a timer expires, a timer event entry is added to the event queue. When `update` is called, handlers for all **queued** events are run. This causes `handler` to run which again schedules another timer to expire immediately and enqueue another event. Thus the event loop invocation runs that as well. On the other hand, the one second timer has not expired as yet and hence is not on the event queue. Since no more event handlers are pending, the event loop terminates and the update command returns. Thus we see `Event 0` and `Event 1` being printed and not `Event 2`. If you do run `update` (or the event loop by any other means) after a second has elapsed, you will see the one second timer handler being run as well.

If the `idletasks` argument is passed to the `update` command, only the idle task queue is processed, not the event queue.

```
after 0 [list puts "After 0"]
after idle [list puts "After idle"]
update idletasks
→ After idle
```

The `after idle` command registers a script to be run when the event loop is idle. Notice from the output that only the idle event handler was run. The timer event handler was not run even though it would have been queued. The example also illustrates that `update idletasks` forces any idle handler to be invoked **even if other events were queued** and the event loop was not actually idle.

Update considered harmful

The `update` command is often used to keep an application's user interface responsive during the course of a long computation. This is not considered good practice due to reentrancy and data consistency issues. The Wiki page https://wiki.tcl-lang.org/1255 discusses this in some detail.

In the author's experience, the only time an `update` has been necessary is for forcing window geometry calculation and propagation in the Tk GUI extension on some platforms that use native OS widgets.

19.2.4. Event handlers and the call stack

It is worthwhile taking a look at how the call stack (Section 14.1) and the C stack (Section 14.1.6) appear when an event handler is running when invoked via `vwait`. Consider the following snippet.

```
proc handler {} {
    puts "handler level: [info level]"
    set ::done 1
}
proc demo {} {
    puts "demo level: [info level]"
    after 0 handler
    vwait ::done
}
demo
→ demo level: 1
  handler level: 1
```

Figure 19.1 shows a snapshot of the call and C stacks while the `handler` procedure is running via the event loop.

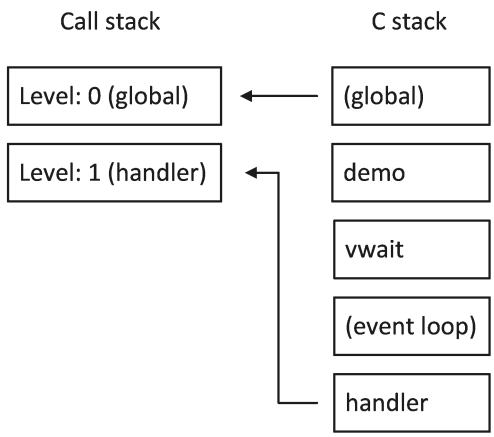

Figure 19.1. Call stack in an event handler

Note the following points from the figure:

- From the perspective of the internal C stack, the call to `vwait`, which in turn runs the event loop and executes the handler, all add a level. Further calls to `vwait` or `update` from within the event handler would add more levels to the C stack.
- On the other hand, the call stack that maintains execution and variable contexts is reset to the global level by the event loop. The output of our little script which showed both `demo` and `handler` executing at level 1 corroborates this. This illustrates that event handlers are run in the global context. You cannot expect to use `upvar` (Section 14.1.4) or `uplevel` (Section 14.1.5) to reach into the context of the `demo` procedure. Note that those contexts are not lost, they will be restored once the `vwait` command returns.

19.3. Scheduling execution of code: after

We now turn our attention to how we can add entries to the event and idle queues. There are several mechanisms through which this can happen. Here we only look at the simplest:

- Timer expiry events which can be used to run code after a specified interval.
- Registering tasks to be run when the system is idle and no events are pending processing.

The `after` command is used for both purposes, and more.

As of Tcl 8.6, one limitation of the `after` command to keep in mind is that it depends on the system clock for time-keeping. If the system clock is inaccurate or jumps for whatever reason, `after` will not correctly measure intervals. There is work in progress to fix this behaviour which will likely show up in future releases of Tcl.

19.3.1. Suspending execution

```
after MILLISECS
```

The `after` command can be used suspend execution, including the event loop, in the current thread for a specified interval. To put the current thread to sleep for a quarter second,

```
after 250
```

19.3.2. Scheduling code

```
after MILLISECS SCRIPT ?SCRIPT …?
```

The `after` command can also schedule a script to run after a specified time interval. The command sets up a timer that will expire after *MILLISECS* milliseconds and returns immediately. The result of the command is an identifier for the timer that can be passed to `after cancel` (Section 19.3.4) to cancel it if desired.

When the timer expires, an entry is added to the event queue with a handler that will invoke the script formed by concatenating the *SCRIPT* arguments separated by a space in the same manner as the `concat` (Section 5.18) or `eval` (Section 3.13) commands.

We have already seen examples of this command for setting up timers. Here is a slightly more realistic example that uses the `http` package to retrieve a Web page in conjunction with a timeout within which the transaction must complete.

```
package require http
proc http_data_sink {token} {
    set ::status done
}
proc geturl_with_timeout {url ms} {
    after $ms {set ::status timeout}
    set http_token [http::geturl $url -command http_data_sink]
    vwait ::status
    if {$::status eq "timeout"} {
        http::cleanup $http_token
        error "Operation timed out."
    }
    set data [http::data $http_token]
    http::cleanup $http_token
    return $data
}
geturl_with_timeout http://www.example.com 10000
→ <!doctype html>
  <html>
  <head>
...Additional lines omitted...
```

That works fine, but if we give it insufficient time to do its work, the timer event fires first and the operation is timed out.

```
% geturl_with_timeout http://www.example.com 10
Ø Operation timed out.
```

As a special case, if *MILLISECS* is `0`, the timer expires immediately and an entry for the handler is appended to the event queue. This idiom is often used when the programmer wants some piece of code to run only after the current handler completes execution. Another use is to break up a long computation into smaller pieces while allowing other handlers to run. We will say more about this in Section 19.3.3.1.

19.3.3. Running on idle: after idle

```
after idle SCRIPT ?SCRIPT …?
```

The `after` command can also add tasks to the idle task queue. The command behaves almost identically to the `after 0` form of the command except that the script formed by concatenating the *SCRIPT* arguments is added to the idle task queue instead of the event

queue. Here is a short example that illustrates the relation between the two queues. Notice the idle handler runs **after** the timer event handler even though it was queued first.

```
% after idle [list puts "Idle task executed"]
→ after#14
% after 0 [list puts "Event handled"]
→ after#15
% update ❶
→ Event handled
  Idle task executed
```

❶ Run all pending event and idle tasks

19.3.3.1. Avoiding event queue starvation

Running a long computation prevents the event loop from running and responding to events. A common technique used to avoid this is to break up the computation into pieces and run each part in turn from the event loop.

We will illustrate this in the context of a previous example of summing the first N natural numbers. Our procedure will do one addition operation and then queue itself back on the event queue to continue with the next stage of computation. Assuming you are running in `tclsh` and not `wish`, where the event loop is already running, we will need to call update to run the event loop.

```
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after 0 [list background_sum [incr n -1] $sum]
    }
}
after 0 background_sum 2
update
→ Calculating...
  Calculating...
  Sum is 3
```

The `update` (Section 19.2.3.2) command keeps executing handlers as long as the event queue is not empty. Since we keep adding a timer (with a `0` expiration) the event loop will keep running until the computation is completed. While this computation is going on, other events that may arrive in the meanwhile will be queued and executed in the order of arrival. The user interface will stay responsive, network connections will be accepted and so forth.

However, there is one flaw in the above. As long as there are entries in the event queue, their handlers will be executed and the **event loop will never move on to the idle task queue**. And since `background_sum` keeps queuing itself, the event queue will never be empty until the computation is completed. The idle task queue will be starved of any execution cycles and activities like updating of windows which happen at idle time will not happen.

To demonstrate this, let us write a script that will run in the background on the idle queue.

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after idle [list idler [incr n -1]]
    }
}
```

Now we fire up the two scripts.

```
after idle idler
after 0 background_sum 2
update
→ Calculating...
  Calculating...
  Sum is 3
  Idle!
  Idle!
  Idle!
```

As shown by the output, the idle task does not get to run until the computation was done.

Queuing our computation on the idle task queue would not solve the problem either because once the event loop starts processing the idle task queue, it will continue to do so until it is empty. We will therefore starve the event queue instead.

The solution to this is to modify our procedures as follows

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after 0 [list after idle [list idler [incr n -1]]] ❶
    }
}
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after idle [list after 0 [list background_sum [incr n -1] $sum]] ❷
    }
}
```

❶ Trampoline from event queue to idle task queue
❷ Trampoline from idle task queue to event queue

Now if we run our previous code, the script output is interleaved indicating neither queue is starved. In essence, instead of rescheduling itself directly, the code "bounces" itself off the other queue ensuring entries on that queue get to run as well.

```
% after idle idler
→ after#25
% after 0 background_sum 2
→ after#26
% update
→ Calculating...
  Idle!
  Calculating...
  Idle!
  Sum is 3
  Idle!
```

This technique applies to computations that directly or indirectly reschedule themselves continuously. If **new** independent events are arriving at a rate faster than the rate at which the queues can be emptied, this would obviously not help.

19.3.4. Cancelling tasks: after cancel

```
after cancel ID
after cancel SCRIPT ?SCRIPT …?
```

Any timer events and idle tasks that have been scheduled with the `after` command can be cancelled with `after cancel` which can take one of two forms.

In the first form, the command's argument is an identifier returned by the `after` or `after idle` commands. The corresponding event is cancelled.

```
set id1 [after 0 puts Timer1] → after#35
set id2 [after 0 puts Timer2] → after#36
after cancel $id1             → (empty)
update                        → Timer2
```

Only the second timer fires as the first one has been cancelled.

In the second form, instead of specifying the timer identifier, the caller can specify the actual script that was scheduled. Repeating the above example but using the idle task queue and the second form of the command,

```
after idle puts Timer1   → after#37
after idle puts Timer2   → after#38
after cancel puts Timer1 → (empty)
update                   → Timer2
```

Again, only the second timer fires as the first was cancelled.

The command does not raise an error if no matching timer is found.

19.3.5. Introspecting after handlers: after info

```
after info ?ID?
```

The `after info` command can be used to query the current event handlers registered with the `after` commands. If no *`ID`* argument is specified, the command returns a list of identifiers of the currently active handlers.

```
after 1000 {puts "Timer"} → after#39
after idle {puts "Idle"}  → after#40
after info                → after#40 after#39
```

If *`ID`* is specified, it returns a pair whose first element is the associated handler script and second is either `timer` or `idle` depending its type.

```
% foreach id [after info] {
    lassign [after info $id] script type
    puts "$id ($type): $script"
    after cancel $id
}
→ after#40 (idle): puts "Idle"
  after#39 (timer): puts "Timer"
```

Note that timers that have already been triggered or that have been canceled do not show up in the results returned by `after info`.

19.4. Event loop error handling

When an error exception is raised it is propagated up the call stack (Section 15.2.2) until it is handled at some call level. If it reaches all the way to the global level in a Tcl application that is not event-driven, the result is application dependent. The `tclsh` shell in interactive mode will trap the error and display it to the user. In non-interactive mode, `tclsh` will by print the error to standard output and exit.

Things work differently with event driven applications. If an error is raised during the execution of an event (or idle task) handler and propagates up to the event loop, it is reported through a background exception handler. The event loop invokes this handler with two additional arguments: the interpreter result and a dictionary of return options. These are exactly the same values that are captured by a `catch` command and are described in Section 15.4.1 and Section 15.4.2.

The `tclsh` and `wish` shells provide their own default background exception handlers which display the error message on the `stderr` (Section 13.2) channel and through a window dialog respectively.

Here is a demonstration of the `tclsh` default background error handler. Notice that `catch` (Section 15.4.1) returned a `0` indicating the `update` command ran without errors. The generated error exception was handled by the event loop and not propagated to `update`. The

default background exception handler invoked by the event loop printed the error stack to the standard error channel.

```
% proc demo {arg} {}
% after 0 demo ❶
→ after#2
% catch update
→ wrong # args: should be "demo arg"
      while executing
  "demo"
      ("after" script)
  0
%
```

❶ Intentionally call `demo` with wrong number of arguments

19.4.1. Custom background error handling: interp bgerror

```
interp bgerror INTERPRETER CMDPREFIX
```

An application can choose to customise default handling of background exceptions by calling `interp bgerror`.

The handling of background errors can be customized on a per-interpreter basis and the *`INTERPRETER`* argument specifies the path of the interpreter to be customized. We will take a deeper look at interpreter paths in Chapter 23 but for the moment we just mention that the empty string refers to the current interpreter.

The *`CMDPREFIX`* argument is a command prefix that will be called with two additional arguments, the error result and return options dictionary.

Let us define our own background exception handler for the above example.

```
% proc bghandler {message ropts} { puts stderr "MyApp error: $message" }
% interp bgerror {} bghandler
→ bghandler
% after 0 demo
→ after#2
% catch update
→ MyApp error: wrong # args: should be "demo arg"
  0
```

Our error handler does essentially the same thing as the default one except adding an application name and only printing the error message instead of the whole stack.

In old versions of Tcl, the global `bgerror` command was used for customizing background error handling. This is now deprecated in favour of `interp bgerror`.

20

Processes and Pipelines

It is a ubiquitous practice in software to start new processes as subtasks for special purposes and exchange data with them via their standard input and output. This chapter describes the facilities Tcl provides for this purpose.

- The `exec` command starts one or more child processes and returns any content written to their standard output as the result of the command.
- The `open` command provides more flexibility by returning a channel that can be used to communicate with child process(es).
- The `tcl::process` command ensemble manages subprocesses.

Both `exec` and `open` also support *process pipelines* wherein multiple processes are chained via their standard input and output.

20.1. Executing child processes: exec

```
exec ?-keepnewline? ?-ignorestderr? ?--? ARG ?ARG …? ?&?
```

The `exec` command starts a *pipeline* of one or more processes. The *`ARG`* arguments specify one or more programs to run along with their parameters or special character sequences that separate the programs and indicate redirection of input and output. If the last argument is not the `&` character, the command result is the data written to standard output by the last process in the pipeline. Unless the `-keepnewline` option is specified, any trailing newline character in the output is discarded. The significance of the `&` character and the `-ignorestderr` option are discussed later. As always, `--` indicates the end of options.

In the simplest case, the command starts a single process and returns the data written to its standard output as the result of the command. For example, here we run the netstat program and collect its output.

```
% set connections [exec netstat -n]
→
  Active Connections

    Proto  Local Address          Foreign Address        State
    TCP    127.0.0.1:55496        127.0.0.1:55497        ESTABLISHED
...Additional lines omitted...
```

In the general form, the arguments can specify multiple programs comprising a process pipeline where each program and its parameters is separated from the other programs by a `|` or `|&` character sequence. In the former case, the standard output of the preceding process in the pipeline is fed into the standard input of the next process. In the latter case, both standard output and standard error of the preceding process are piped.

In the absence of any I/O redirection or errors, the output of the last process in the pipeline is returned as the result of the `exec` command.

Here is a pipeline where we filter the output of `netstat` through the `findstr` program on Windows to only retrieve UDP connections.

```
% set udp_connections [exec netstat -an | findstr UDP]
→   UDP    0.0.0.0:53             *:*
    UDP    0.0.0.0:3702           *:*
    UDP    0.0.0.0:3702           *:*
...Additional lines omitted...
```

20.1.1. Passing program arguments

When passing argument values to the executed programs, keep in mind that the arguments first undergo substitutions as per Tcl's quoting rules (Section 3.2). Depending on the program being executed, they may then be subject to that program's quoting rules, which in all likelihood differ from those of Tcl. Thus care must be taken to appropriately escape program arguments when special characters are part of the passed arguments. Unfortunately, different programs follow different conventions, particularly on Windows, so the escaping of special characters is necessarily program-specific.

The same also applies to file paths that may be passed to the child processes. On Windows for example, many programs do not accept `/` as a path separator. Thus attempting to produce a directory listing of the Tcl installation binary directory with Windows command shell's `dir` internal command will fail.

```
% exec cmd /c dir [file dirname [info nameofexecutable]]
Ø Parameter format not correct - "l".
```

The passed file path must be transformed into native form with the `file nativename` command (Section 12.1.7).

```
% exec cmd /c dir [file nativename [file dirname [info nameofexecutable]]]
→  Volume in drive C is OS
   Volume Serial Number is 762B-8D11

   Directory of c:\tcl\magic\bin

  01/04/2025  08:26 PM    <DIR>          .
  01/04/2025  08:26 PM    <DIR>          ..
...Additional lines omitted...
```

This transformation of file paths is only required for program arguments. If a directory path is specified for the program name, Tcl will automatically convert it to the native form.

Also to be noted is that Tcl's `exec` behaviour should not to be confused with that of Unix shells. The latter implicitly do glob-style expansion of wildcard patterns in arguments to programs whereas Tcl's `exec` command does not. Thus the following command executed in a Unix shell

```
ls *.c
```

would not be

```
exec ls *.c
```

in Tcl but rather

```
exec ls {*}[glob *.c]
```

20.1.2. Locating programs

The program to be executed for each stage of the `exec` pipeline may be an executable image or a shell script (on Unix) or a batch file (on Windows). It may be specified as an absolute path, a relative path or just a file name with no directory component.

The path will undergo tilde substitution in Tcl versions prior to 9.0.

If the program is specified purely by name (i.e. no directory components are present), `exec` will look for the file in an operating system dependent fashion. On Unix, it looks for the file in directories specified in the `PATH` environment variable. On Windows, the command will look in the directory containing the Tcl application, the current directory, the Windows system directories and finally the directories in the `PATH` environment variable.

20.1.2.1. Locating shell internal commands: auto_execok

```
auto_execok PROGRAMNAME
```

Some "programs" that are executed in command shells are not separate executables at all but are actually implemented internal to the shell. For example, the `DIR` command for listing directories at the Windows command prompt is internal to the Windows `cmd.exe` shell. Attempting to run this directly from Tcl will raise an error.

```
% exec dir *.*
→ couldn't execute "dir": no such file or directory
```

Tcl provides a command, `auto_execok`, for dealing with such commonly used commands that are built into the operating system command shells. The command returns a list of words to be passed to `exec` (Section 20.1) to run that program.

```
auto_execok notepad → C:/WINDOWS/system32/notepad.EXE
auto_execok dir     → C:/Windows/System32/cmd.exe /c dir
```

Notice the difference in the two outputs. Since `dir` is an internal command of `cmd.exe`, `auto_exec` returns the command prefix to be used to invoke it. We can then run it as

```
% exec {*}[auto_execok dir] *.*
→  Volume in drive C is OS
   Volume Serial Number is 762B-8D11

   Directory of C:\TEMP\book

...Additional lines omitted...
```

Additionally, on Windows platforms if the search fails and no extension was specified in the program path, the command will repeat the search by appending the extensions in the `PATHEXT` environment to the file name.

20.1.3. Redirecting I/O

By default, the output of each process in the pipeline is supplied as the input to the next process. This behaviour can be changed for each process in the pipeline for both input and output through special character sequences in the arguments to `exec`. This I/O redirection takes a form very similar to that used in Unix or Windows command shells.

20.1.3.1. Redirecting input

By default, the **first** process in an `exec` pipeline reads its standard input from the standard output of the parent Tcl application that invoked the `exec` command. This behaviour can be changed so that the process gets its input

- from a file
- from an open channel in the Tcl application
- from a value in the Tcl application

We will demonstrate all three techniques by recursively invoking `tclsh` as a separate process and printing its PID.

On Windows platforms, the examples will not work with the `wish` shell because GUI programs on Windows do not have a real operating system provided standard input or output.

Redirecting input from a file

To have the first process read its standard input from a file, prefix the file path with a `<` character. Let us first write out our sample file that we use as input.

```
set chan [file tempfile temppath]
puts $chan { puts "My PID is [pid]" }
close $chan
```

Now to recursively invoke ourselves with standard input for the child process redirected to this file, pass the file path prefixed with `<` to the `exec` command.

```
exec [info nameofexecutable] <$temppath → My PID is 23968
```

Note that you can optionally separate the file path from the `<` character with whitespace so we could also have written the above as

```
exec [info nameofexecutable] < $temppath → My PID is 22092
```

(Note the space character before the `temppath` reference.)

Redirecting input from a channel

As an alternative to redirecting input from a file, you can redirect input from a channel that is already open by prefixing the channel with the `<@` character sequence. For example, a variation of the above example:

```
set chan [open $temppath r]          → file24154ad40a0
exec [info nameofexecutable] <@$chan → My PID is 19140
close $chan                          → (empty)
```

As shown in the example, the channel must have been opened for reading for channel based redirection to work. As before, the child process will exit when it encounters an EOF on the input channel.

Not all channel types are supported for use with input redirection. In particular, on Windows platforms network socket based channels cannot be used unlike on Unix. On all platforms reflected channels (Section 21.3) also do not work with input redirection.

There are two factors in mind when using the channel based input redirection. The first is that the file access pointer is shared between the parent and the child so the following sequence of commands will not yield the desired result.

```
set fd [open foo.txt w+]           → file24154ad3520
puts $fd {puts "My PID is [pid]"}  → (empty)
exec [info nameofexecutable] <@$fd → (empty)
close $fd                          → (empty)
```

The reason this does not work is that after the write to the channel, the file access pointer is positioned at the end. Consequently, when the child process reads from the channel it only sees the end-of-file marker and exits. This problem can be fixed by inserting a seek to reposition the access pointer to the front of the file.

```
set fd [open foo.txt w+]           → file241549e3f10
puts $fd {puts "My PID is [pid]"}  → (empty)
chan seek $fd 0 start              → (empty)
exec [info nameofexecutable] <@$fd → My PID is 19928
close $fd                          → (empty)
```

Now the child process reads the file as desired.

The other point to remember is the potential for race conditions. If you change the order of operations to invoke the `exec` before the `puts`, it is possible that the child process will run before the write to the file, find it empty and exit.

For these reasons, the channel redirection mechanism is not recommended as a means for continuous communication between the parent and the child. We will see alternative means described in Section 20.2 and Section 20.3 that are more suitable for such scenarios.

Redirecting input from a Tcl value

The final option available for redirecting the standard input of the spawned process is through the `<<` redirection operator. Instead of a file path or a channel, this redirects input from the specified value. Our example could be written as

```
% exec [info nameofexecutable] << {puts "My PID is [pid]"}
→ My PID is 24320
```

Tcl arranges for the argument following the `<<` to be passed in to the child process in its standard input. We have specified the value in our example as a braced string literal. Of course it could also have been the result of a command or a variable reference as well.

20.1.3.2. Redirecting output

Just as for the input side, standard output and error of processes in an `exec` pipeline can also be redirected. There are more combinations possible here which can be confusing so we first lay out elements that are common to all.

- Only the standard output of the **last** process in the pipeline can be redirected. Other processes always send their output to the next process.
- On the other hand, the redirection of standard error output applies to **all** processes. It is not possible to only redirect it for any single process (not even the last in the pipeline).
- The single `>` character sequence always indicates writing to the channel at its current file access position. The double `>>` sequence always appends.
- A prefix of `2` before any of the above character sequences indicates the redirection applies to standard error and not to standard output.
- A suffix of `&` indicates the redirection applies to both standard output and standard error.
- The `@` character specifies redirection to an open channel in the Tcl application calling the `exec`. The channel must have been opened for writing.
- If output is redirected, the result of the `exec` command is the empty string.

Sending standard output to a file

The `>` character followed by a file path will write the standard output of the last process in the pipeline to that file, overwriting it if it already exists.

```
exec netstat -an | findstr ESTABLISHED > connections.log
```

Note that the result of the `exec` command above is the empty string due to the redirection.

The `>>` redirection works similarly except that it will **append** to the file.

Sending standard error to a file

The `2>` character sequence followed by a file path will write the standard error of all processes in the pipeline to that file, overwriting it if it already exists. The standard outputs are unaffected. In the example below we spawn off another Tcl interpreter and have it print messages to standard output and error.

```
% exec [info nameofexecutable] 2> error.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
→ This is standard output
% readFile error.log
→ This is standard error
```

As seen above, only the standard error is redirected to the file while the standard output appears as the result of the `exec` command.

The standard error output can be appended to the file instead of overwriting it by using the `2>>` redirection instead.

Redirecting the standard error modifies the error handling behaviour of `exec`. This is discussed in Section 20.1.4.

Sending output to a channel

Instead of sending standard output and error to a file, it can be sent to an open channel in the current Tcl interpreter by using `>@` and `2>@` respectively.

```
% set chan [file tempfile temppath]
→ file241563ca200
% exec {*}[auto_execok date] /t >@ $chan
% close $chan
% readFile $temppath
→ Thu 04/03/2025
```

Note that there are no equivalent redirections that append to channels.

Conflating standard output and error

The redirections discussed so far have dealt with independently redirecting standard output and error. For example, one might write

```
exec grep Tcl_.*Init {*}[glob *.c] > matches.txt 2> errors.txt
```

In cases where you want both standard output and error to go to the **same** destination, append a `&` character to the appropriate operator. Thus `>&` and `>>&` will overwrite or append the standard output of the **last** process and the standard error of **all** processes in the pipeline to the specified file.

```
exec [info nameofexecutable] >& output.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
readFile output.log
→ This is standard output
  This is standard error
```

The `>&@` works similarly except it writes to an open channel.

The use of the `>&` as above is **not** the same as separately redirecting the standard output and error to the same file as below.

```
exec [info nameofexecutable] > output.log 2> output.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
readFile output.log
→ This is standard error
```

As you see from the above, the output from one can overwrite or mangle output from the other.

One final form, `2>@1`, redirects standard error to be included as part of the command result. Since the standard output is already included in the result, the following will return both as the result of the `exec` command.

```
set result [exec [info nameofexecutable] << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
} 2>@1]
puts $result
→ This is standard output
  This is standard error
```

20.1.4. Error handling in exec

Error handling for `exec` invocation is complicated by the fact that the error may come from different sources:

- The `exec` command itself may raise an error if one of the programs in the pipeline is not found or does not have execute permission for the user and so on.
- The program(s) run but terminate with some error condition.

Moreover, the executed programs may signal error conditions or abnormal termination in different ways:

- The process may exit with a non-0 exit code.
- The process may write to its standard error output.

The application needs to be able to recognize and distinguish these different forms. A further complication is that not all programs follow the above conventions. A search application may use the exit code as a result value returning the number of matches. Others may use standard error to record progress messages, not necessarily errors. For these reasons, error detection is very much dependent on the program(s) being executed. The discussion below is focused on distinguishing the various cases. Interpretation as errors or normal behaviour is up to the application.

We can interactively explore various scenarios by spawning a Tcl process that imitates application behaviour. For starters we assume standard error has **not** been redirected as that affects error handling.

First, consider an attempt to execute a program that does not exist.

```
% catch {exec nosuchprogram} result ropts
→ 1
% dict get $ropts -errorcode
→ POSIX ENOENT {no such file or directory}
```

As expected, the `catch` (Section 15.4.1) command result indicates an error. The `-errorcode` (Section 15.4.2.3) element of the return options dictionary gives the details of the failure. Other error codes are also possible, such as insufficient permissions. These are generally returned as POSIX error codes.

Another possibility is that the program starts up but suffers abnormal termination via a segment violation, signal etc. In this case the error code will be of the form

```
CHILDKILLED PID SIGNALNAME MESSAGE
```

where *`PID`* is the process identifier of the terminated child process, *`SIGNALNAME`* indicates the signal (`SIGTERM`, `SIGSEGV` etc.) that forced the termination and *`MESSAGE`* is the human readable description of the reason for termination. For example, a null pointer access in the child process would result in an error code of

```
CHILDKILLED 12408 SIGSEGV {segmentation violation}
```

Finally, there is the possibility of the program itself signalling an error. Tcl considers a child process exiting with a non-0 exit code **or** writing to its standard error output to be an error. We can simulate both these conditions by recursively invoking the Tcl shell.

```
catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        exit 3
    }
} result ropts
→ 1
```

The `catch` command (Section 15.4.1) returns an error because child exited with a non-0 exit code. The error code from the return options dictionary also includes this exit code. Just as for the normal completion of an executed program, the result includes its standard output. In addition, the error message is also appended to this result.

```
% puts [dict get $ropts -errorcode]
→ CHILDSTATUS 16376 3
% puts $result
→ This is standard output
  child process exited abnormally
```

The other situation considered as an error is if the child writes to its standard error. This is simulated by the following snippet. Again, the `catch` command result indicates an error exception even though the child process exited with an exit code of `0`. This is because it wrote to its standard error. The error code however shows up as `NONE`.

```
% catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
        puts "This is standard output again"
        exit 0
    }
} result ropts
→ 1
% puts [dict get $ropts -errorcode]
→ NONE
```

The result of the exception includes the standard output of the child followed by its standard error content. Note the latter always appears at the end no matter the order in which `puts` statements were executed.

```
% puts $result
→ This is standard output
  This is standard output again
  This is standard error
```

If `-ignorestderr` is specified, `exec` does not treat output to standard error as an error.

```
% catch {
    exec -ignorestderr -- [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
    }
} result ropts
→ 0
% puts $result
→ This is standard output
```

The result now includes only standard output and the command does not raise an error.

Another way to accomplish the same thing is to redirect the standard error using any of the error redirectors like `2>`.

The following pseudocode template summarizes handling of all these various cases using the `try` command Section 15.4.3. Depending on the program being executed, the application can take appropriate action depending on whether the condition signifies a real error or not. Errors for which trap clauses are not specified will be automatically propagated.

```
try {
    set result [exec command parameters ...]
} trap NONE output {
    # Application exited with a 0 exit code but wrote to standard error.
    # The variable output will contain the standard output content
    # followed by the standard error content
    ...do whatever...
} trap CHILDSTATUS {- ropts} {
    # Child exited with a non-0 exit code
    # Retrieve the PID and exit code
    lassign [dict get $ropts -errorcode] -> pid exit_code
    ...do whatever...
} trap CHILDKILLED {- ropts} {
    # Child terminated abnormally
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
} trap CHILDSUSP {_ ropts} {
    # Child suspended
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
} trap POSIX {- ropts} {
    # Other errors like permissions, file not existing etc.
    # Retrieve POSIX error mnemonic and reason
    lassign [dict get $ropts -errorcode] -> posix_code reason
    ...do whatever...
}
```

20.1.5. Running background processes

Normally the `exec` command waits for all processes in the pipeline to terminate and returns as its result the standard output of the last process in the pipeline. However, if the last argument to `exec` is `&`, the command returns immediately running the processes in the background. The return value is a list containing the process identifier (PID) of each process in the created pipeline.

In the following example, the `exec` command returns immediately without waiting for the `netstat` and `findstr` processes to finish executing.

```
% exec netstat -an | findstr ESTABLISHED > connections.log &
→ 660 17736
```

The list of PID's returned is in the order of processes in the pipeline.

If no I/O redirection is in effect, the standard output and standard error of the last process in the pipeline will go to the application's standard output and error.

20.1.6. Limitations in exec

There are certain limitations in `exec` for some scenarios:

- It does not provide for an "interactive" bidirectional data exchange with the child process. For example, we cannot use it fire up another copy of our Tcl applications and feed it a command, get back the result and then repeat that sequence.
- There is no control over encodings, line translations etc. with respect to data written to and read from the child process. The encoding used is always the system encoding as returned by the `encoding system` command.
- Its syntax makes it not just difficult but impossible to pass arguments that match certain character sequences such as those used for redirection.
- It has certain platform-specific limitations such as not being able to execute child process with elevated privileges on Windows.

The first two of these are addressed in the next section and a proposal to fix the third is in place for a future release of Tcl. To overcome the last limitation, you will need the help of third party extensions.

20.2. Channels for process pipelines: open

```
open PATH ?ACCESS? ?PERMISSIONS?
```

We introduced the `open` command in Section 13.3 where we used it to create an I/O channel to a file on disk. The command is in fact more general in that it can be used to create channels of different types. Here we examine its use for creating process pipelines.

Use of `open` to create channels for process pipelines has several advantages over `exec` at the cost of some slight complexity.

- It permits arbitrary sequences of data exchange with the child process.
- The data exchange can be asynchronous, a capability we will describe in Section 21.1.
- Being a channel, we are able to use all capabilities of Tcl channels including appropriate encodings, channel transforms (Section 21.2) etc. For example, we could transparently compress the data we are piping into the process pipeline.

A channel to a process pipeline is opened using the same syntax we saw in Section 13.3.

The *ACCESS* and *PERMISSIONS* arguments are the same as described there for file channels. The *PATH* argument on the other hand must begin with the | character. The rest of the *PATH* argument is treated in the same fashion as the arguments to exec (Section 20.1).

Used in this form, open returns a channel that may be used to write to the standard input of the first process in the pipeline or read from the standard output of the last process (assuming no redirections are in effect). The returned channel must as always be released with the close command when done.

If the channel to a process pipeline is in blocking mode, the close will not return until all processes in the pipeline have ended.

The operations permitted on the returned channel depend on the *ACCESS* argument as shown in Table 20.1. The descriptions in the table assume that no redirections are in effect. For example, if the output of the pipeline is redirected to a file, no data will be read from the channel. Some illustrative examples follow the table.

Table 20.1. Access mode for pipelines using open

Mode	Description
r, rb	The channel is opened as read-only in text and binary modes (Section 13.14) respectively. The standard input of the first process in the pipeline is taken from the standard input of the current process. The standard output of the last process in the pipeline can be read from the created channel.
r+, rb+, r+b, w+, wb+, w+b	The channel is opened for reading and writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The standard output of the last process in the pipeline can be read from the created channel.
w, wb	The channel is opened only for writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The channel cannot be read and any output from the last process in the pipeline will go to the current standard output unless any redirection is in place.

A read-only pipe

Our first example is a variation of one of the exec based ones we saw earlier. We list network connections using the netstat program and pipe the output to findstr to filter them. Since we never need to pass any data **to** the process pipeline, we can open the channel in read-only mode.

```
% set chan [open "|netstat -an | findstr ESTABLISHED" r]
→ file24154b96a30
% while {[gets $chan line] >= 0} {
    puts $line
}
→   TCP    127.0.0.1:55496        127.0.0.1:55497        ESTABLISHED
    TCP    127.0.0.1:55497        127.0.0.1:55496        ESTABLISHED
...Additional lines omitted...
% close $chan
```

Having a channel in hand, we can process the output data a line at a time should we choose unlike for `exec` where we got all the data in one lump. For our example, this may not matter much since the output data is limited in size and easy enough to split into lines. But in the general case, where the data is either very large or an continuous stream (we will see an example of this later), `exec` is not a viable option.

A write-only pipe

Our second example involves a write-only pipe where we will write to the `gzip` program to compress data that we will generate incrementally.

```
% set chan [open "|gzip - > foo.zip" w] ❶
→ file7
% chan configure $chan -encoding utf-8 -translation lf
% puts $chan "Line 1"
% puts $chan "Line 2"
% close $chan
```

❶ Specifying `-` as the input file causes `gzip` to read data from its standard input

Several additional points are illustrated by this example:

- Redirection operators like `>` above can be used with `open` as with `exec`.
- We can call `chan configure` to set various options on the channel. Because most compression programs expect binary data, we configure the channel to transform our Unicode strings to binary data (Section 13.12). Without this, the channel would use system encoding which may or may not be able to handle all characters.
- We do not need to collect all data and feed it to `gzip` in one step as for `exec`. We can write it piecemeal as it is generated.
- We need to close the channel when done. Otherwise, not only will we have a resource leak with the channel handle, it will also cause `gzip` will hang around waiting for more data.

20.2.1. Running tclsh in a pipeline

It is sometimes useful in real world applications to “drive” `tclsh` in a pipeline for executing ancillary tasks, parallelizing computation and so on. We demonstrate such usage here.

Additionally, this will also serve as an example of

- Using a bi-directional pipe where the application both writes to and reads from the child process.

- Using the `list` command to construct the program and argument parameters.
- Additional channel configuration that must be set up for some applications.

If on Windows, the code below must be run from `tclsh` or some other Tcl console application, not from `wish` or a GUI based one as the latter do not have standard input/output.

Our opening command itself looks different from what we have seen earlier.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file24154b96a30
```

The `r+` argument opens the pipe for both reading and writing. The `-encoding` option to `tclsh` informs that it should expect UTF-8 encoded data in its standard input. The list command is used to correctly form the arguments to the `open` command.

Consider if we had written the command as

```
set chan [open "|[info nameofexecutable] -encoding utf-8" r+]
```

Now, if our `tclsh` was installed in a directory with spaces in its path, say under `Program Files` on Windows, Tcl will attempt to execute the following after command substitution.

```
set chan [open "|C:/Program Files/Tcl/bin/tclsh.exe -encoding utf-8" r+]
```

This will fail as space in the path will cause the `open` command to treat `C:/Program` as the name of the program to run. Although some combination of quoting and escapes would also work, it is generally simpler and less error prone to use `list` to correctly form the arguments when command or variable substitution are involved as in our example.

As an aside, we could have placed the first argument in quotes

```
set chan [open "|[list [info nameofexecutable] -encoding utf-8]" r+]
```

but that is not necessary as long there is no whitespace between the `|` and `[` characters.

The next thing you need to be aware of relates to buffering in the channel (Section 13.6.1). By default, the created channel is fully buffered as we can verify:

```
chan configure $chan -buffering → full
```

Fully buffered channels offer highest performance but are not convenient in scenarios like ours where commands we write to our child `tclsh` process need to be immediately sent across. We could do this by explicitly calling `chan flush` (Section 13.6.1.2) after every write but it is easier to just set the channel to be line buffered instead. At the same time we also set our channel to use UTF-8 encoding to match the encoding our child `tclsh` is expecting.

```
% chan configure $chan -buffering line -encoding utf-8
```

The child `tclsh` process is now running and since it was not passed a script file on the command line, it will loop reading commands from its standard input, i.e. the pipe, and executing them.

First, let us configure its buffering for the same reason listed above for our side of pipe. We write the appropriate command to the pipe.

```
% puts $chan {chan configure stdout -buffering line}
```

The child `tclsh` will then read the command and set its `stdout` channel configuration appropriately.

At this point, we list two important distinctions between running `tclsh` in a pipeline versus running it interactively.

- In interactive mode, `tclsh` will write a prompt to the standard output when it is ready for the next command. It does not do this when reading from a pipe.
- Secondly, the result of the evaluated command is **not** written to standard output.

We can check whether the child `tclsh` thinks it is in interactive mode by asking it to print the value of the `tcl_interactive` global variable in the child process. This value can then be read back from our end of the pipe.

```
% puts $chan {puts $tcl_interactive}
% gets $chan
→ 0
```

We now know that the child is non-interactive mode. We do not therefore have to worry about dealing with the `tclsh` prompt characters being read from the pipe and having to separate them from the actual data.

At the same time, we need to be aware that in non-interactive mode `tclsh` will not print the result of evaluated commands to its standard output. So if, instead of forcing an explicit write using `puts` as above, we had invoked the following commands

```
puts $chan {set tcl_interactive}
gets $chan
```

our shell would have appeared to hang. The child `tclsh` does not write the result of the `set` to its standard output. Consequently, our `gets` command would sit there waiting forever (since we have not discussed non-blocking I/O as yet).

If for some reason, you want the child `tclsh` to output prompts and display the result of evaluated commands, you can set the value of the `tcl_interactive` variable to `1` (obviously in the **child** `tclsh`, not in our parent shell).

When we are done with the child `tclsh`, we can either send it an `exit` command or simply close our end of the pipe which will cause it to exit. Here we will explicitly ask it to exit.

```
puts $chan {exit} → (empty)
```

Now our attempt to read from the pipe returns an empty string and `eof` (Section 13.7.3) indicates an end of file on the channel at which point we can close it.

```
gets $chan  → (empty)
eof $chan   → 1
close $chan → (empty)
```

20.2.2. Pipeline process ids: pid

```
pid CHANNEL
```

The command returns the list of process identifiers of all processes present in a pipeline associated with a channel and an empty list if *`CHANNEL`* is not a process pipeline.

```
set chan [open "|netstat -an | findstr ESTABLISHED" r] → file24154bc5fe0
pid $chan                                              → 1940 18856
```

20.2.3. Error handling in pipelines

If any of the processes running in the pipeline signal an error either with the exit status or by writing to their standard error as described in Section 20.1.4, the `close` on the pipeline channel will throw an exception.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file241563b8580
% chan configure $chan -buffering line -encoding utf-8
% puts $chan {exit 2} ❶
% close $chan
Ø child process exited abnormally
% puts $::errorCode
→ CHILDSTATUS 22892 2
```

❶ Force child to exit with an error code

As seen above, the error code corresponds to the child exiting with a non-0 status. It does not however indicate which process(es) failed in the case of a multi-process pipeline. See Section 20.6.2 for a more informative alternative available in Tcl 9 but not 8.6.

20.3. Standalone pipes: chan pipe

```
chan pipe
```

In the previous sections, we have seen various ways of creating and communicating with child processes through pipes and redirections. However there are some scenarios that are at best awkward to program for using these means.

One of these involves reading the standard output and standard error of a child process as separate data streams. This is not possible with the redirection forms we have seen where we can at most redirect the standard error into standard output and then somehow separate out the two, which may or may not be possible. Alternatively, standard error may be redirected to a file but the semantics of a file are very different when it comes to EOF and other aspects.

The `chan pipe` command provides a solution.

The command creates an operating system pipe and returns a list containing two channels, the first for reading from the pipe, and the second for writing to it. Any data written to the second channel will be read from the first.

Let us interactively observe how the channels work. First we create the pipe and assign the read and write channels to `rchan` and `wchan` respectively.

```
% lassign [chan pipe] rchan wchan
```

We will be sending simple text across so we configure them to be line buffered so writing a line will immediately flush the pipe.

```
% chan configure $rchan -buffering line
% chan configure $wchan -buffering line
```

Now we write data into the pipe via the write-side channel. As expected, it can be read from the read-side channel.

```
% puts $wchan "Testing 1 2 3..."
% gets $rchan
→ Testing 1 2 3...
```

The two channels are read-only and write-only respectively.

```
% puts $rchan "Fail"
Ø channel "file241546aa8b0" wasn't opened for writing
% gets $wchan
Ø channel "file2415439ded0" wasn't opened for reading
```

Closing the write side channel will be detected as end-of-file on the read side.

```
close $wchan    → (empty)
gets $rchan     → (empty) ❶
chan eof $rchan → 1
close $rchan    → (empty)
```

❶ Empty string returned because of EOF.

Separating standard output and error

Having seen the basic working of a pipe, let us see how it might serve our purpose. We will spawn a second copy of our Tcl shell as the child process.

Again, we start by creating a pipe and then, as before, we use redirection into a channel to have the child process write its standard error to this pipe (Section 20.1.3.2).

```
lassign [chan pipe] rchan wchan
set chan [open |[list [info nameofexecutable] 2>@ $wchan << {
    puts "This is standard output"
    puts stderr "This is standard error"
}]]
close $wchan
```

The channel returned by `open` will be attached to the child's standard output. The write side of our pipe will be attached to the child's standard error through the `2>@` redirection. **We then immediately close the write-side in our application.** This is important because otherwise the write-side pipe file descriptor will have two references — one held by us and the other by the child process. When the child closes its end or exits, we will not see EOF on our read-side channel because the write-side will still be open due to the reference we hold. By closing it, we ensure that we see the EOF when the child closes the write side of the pipe.

Now we can independently read the child standard output and error.

```
gets $chan   → This is standard output
gets $rchan  → This is standard error
close $chan  → (empty)
close $rchan → (empty)
```

Filter programs

Another use of channel pipes arises in conjunction with programs that act as "filters". They read in their standard input, apply some transform to it and write the result to their standard output. We saw examples earlier in this chapter that used `findstr` as a filter.

However, the technique demonstrated there would not work with some filter programs because they do not write their output until they see an EOF on their standard input. For such programs, we are in something of a Catch-22. It will not return any data until sees an EOF. For that to happen, we have to close the channel returned by `open`. But then once we close the channel, we cannot read back the data the program writes!

One solution is to use `open` instead of `exec` and half-close the write side of the channel. Standalone pipes provide an alternate solution usable with `exec` as demonstrated below. We use the `filter_upcase.tcl` script to simulate these filters.

```
# filter_upcase.tcl
fconfigure stdin -buffering line
fconfigure stdout -buffering line
set result ""
while {[gets stdin line] >= 0} {
    append result [string toupper $line]\n
}
puts stdout $result
exit 0
```

The above filter program will keep reading standard input until EOF and then write the upper case form of the input data to its standard output.

Now we need to call this script as a child process, feed it some data and read back the results. Unlike for `open` which returns a channel, here we will need to create **two** pipes; one to send data to the child as its standard input, and one to read data from the child as its standard output.

```
% lassign [chan pipe] childread mywrite ❶
% lassign [chan pipe] myread childwrite ❷
```

❶ Child's standard input
❷ Child's standard output

We then fire up the child with appropriate redirections. We also close the child's end of the pipe within the parent for reasons explained in the previous example.

```
% exec [info nameofexecutable] scripts/filter_upcase.tcl <@ $childread >@ \
    $childwrite &
→ 18156
% close $childread
% close $childwrite
```

We now write to the channel connected to the child's standard input.

```
% puts $mywrite "This is a test"
% puts $mywrite "This is only a test"
```

Then we close our write-side pipe so that the child will see EOF on its standard input and know it can stop reading and write out the transformed data.

```
% close $mywrite
```

The transformed data now appears on our side of the child's standard output channel.

```
% read $myread
→ THIS IS A TEST
  THIS IS ONLY A TEST

% close $myread
```

Now the dirty little secret is that we only used this example as a means of demonstrating pipe usage. Our simple example could have been done more easily using the half-close method discussed in the next section.

20.4. Half-closing of channels

```
close CHANNEL ?DIRECTION?
chan close CHANNEL ?DIRECTION?
```

We described in the previous section one method of running filter programs that wait for EOF on standard input before writing to standard output. The `chan close` command provides an alternate, possibly simpler, means for working with such programs.

We described the basic operation of `chan close` in Section 13.4. As described there, the command can take an optional second argument, `read` or `write`, that specifies that the channel is to be closed only for that direction of data transfer.

We can make use of this capability to close the standard input of the child process. The code below makes use of this technique to run our example child process from the previous section.

We open a pipe to the child process for read and write. We then write our data to the pipe and then close **only the write side** of the pipe which then results in the standard input of the child seeing an EOF.

```
set chan [open |[list [info nameofexecutable] scripts/filter_upcase.tcl] r+]
puts $chan "This is a test"
puts $chan "This is only a test"
chan close $chan write
```

The child will then write to its standard output and since the read side of the channel is still open, we can read its output.

```
% read $chan
→ THIS IS A TEST
  THIS IS ONLY A TEST

% close $chan
```

This technique is clearly simpler, but less general, than the one described in Section 20.3.

Half-closing of channels is not limited to pipes. You can also use it for network sockets where closing the client socket for writes would indicate to the server that no more data is forthcoming from the client while leaving the connection open in the reverse direction.

20.5. Passing environment to child processes

The process environment variables in a child process are inherited from its parent. If you want to pass a different environment to the child, you need to save the `env` array, modify it, start the child process and then restore `env` from the saved copy.

This is further complicated in a multi-threaded environment since the `env` global values are reflected across all threads in a process.

One possible work-around is to run the child process through an intermediary that allows setting of the environment such as `/usr/bin/env` on Unix or the command shell on Windows.

20.6. Managing child processes: tcl::process

The `tcl::process` command ensemble implements monitoring of status and errors in child processes created with `exec` and `open`.

The `tcl::process` command is not available in Tcl 8.6 and earlier.

20.6.1. Enumerating subprocesses: tcl::process list

```
tcl::process list
```

The `tcl::process list` command returns processes spawned by `exec` or `open`.

```
set first [exec [info name] nosuchfile.exe &]  → 16216
set second [exec [info name] nosuchfile.exe &] → 6304
tcl::process list                              → 16216 6304
```

20.6.2. Checking child process status: tcl::process status

```
tcl::process status ?-wait? ?--? ?PIDLIST?
```

When `exec` is invoked asynchronously, it immediately returns the child process' PID. Any errors in the child process do not result in a Tcl exception being thrown. In these cases, the `tcl::process status` command can be used to determine the child's status.

`PIDLIST` is an optional argument containing a list of child process ids. If the argument is not passed, the command returns the status of all child processes.

The result is a dictionary mapping each PID to a status value. If the child process is still running, the status is an empty list. Otherwise it is a list of up to three elements. The first is the exit code of the process. A non-zero exit code signifies an error in which case the second element is an error message and the third an error code list in the standard `errorCode` format. Any PID's in *`PIDLIST`* that are not child processes are not included in the returned list.

```
% set child [exec [info nameofexecutable] nosuchfile.tcl &]
→ 19796
% after 100 ❶
% tcl::process status $child
→ 19796 {1 {child process exited abnormally} {CHILDSTATUS 19796 1}}
```

❶ Wait for process to exit.

Normally, the `tcl::process status` command returns immediately. However, the `-wait` option can be specified to have it block until all processes passed in the argument have exited, either normally or with error. This can also be used to synchronize with one or more asynchronously started processes.

```
% set fd [open "|[info nameofexecutable] | [info nameofexecutable]" r+]
→ file241546aa9b0
% set children [pid $fd]
→ 15588 5124
% tcl::process status $children ❶
→ 15588 {} 5124 {}
% puts $fd "puts {exit 0}; flush stdout; exit 1" ❷
% flush $fd
% print_dict [tcl::process status -wait $children] ❸
→ 5124    = 0
  15588   = 1 {child process exited abnormally} {CHILDSTATUS 15588 1}
% close $fd ❹
Ø child process exited abnormally
```

❶ Empty status as processes still running.
❷ First process exits with error, second with success.
❸ Wait for both child processes to exit.
❹ Will raise an error as the first process, which is tied to the channel exited with an error.

20.6.3. Cleaning up process resources: tcl::process purge|autopurge

```
tcl::process purge ?PIDLIST?
tcl::process autopurge ?BOOL?
```

When a process started by Tcl exits, certain resources are not immediately released until the `waitpid` system call is called (on Unix) or the subprocess handle is closed (on Windows). By default, this is done on the next call that starts a subprocess.

For cases where additional control is desired, Tcl provides two commands, `tcl::process purge` and `tcl::process autopurge`.

The `tcl::process purge` command cleans up the resources associated with the specified child processes or all child processes if the *`PIDLIST`* argument is not passed. Once this is done, the exit status of those subprocesses is no longer available. This has multiple benefits in that the resources are not left hanging around until the next `exec` and provides control over which resources are released.

The other related command, `tcl::process autopurge`, provides control over Tcl's automatic purging on every `exec` or `open` pipeline invocation. Without arguments, the command returns the current setting of automatic purging. If called with a boolean, this automatic purging is set accordingly. Turning off autopurging allows several subprocess invocations to be done in sequence without losing the exit status of preceding subprocesses.

```
% tcl::process autopurge 0
→ 0
% set first [exec [info name] nosuchfile.exe &]
→ 22308
% set second [exec [info name] nosuchfile.exe &]
→ 20436
% after 100 ❶
% print_dict [tcl::process status]
→ 20436   = 1 {child process exited abnormally} {CHILDSTATUS 20436 1}
  22308   = 1 {child process exited abnormally} {CHILDSTATUS 22308 1}
% tcl::process purge $first ❷
% print_dict [tcl::process status]
→ 20436   = 1 {child process exited abnormally} {CHILDSTATUS 20436 1}
% tcl::process purge ❸
% tcl::process status
```

❶ Wait for processes to exit
❷ Status of `$first` is purged, `$second` still available.
❸ ALL subprocess status is discarded.

21

Advanced I/O

In Chapter 12 we introduced the channel abstraction and basic operations in Tcl in the context of reading and writing files. We also saw the use of channels for I/O with child process pipelines in Section 20.2. We now expand on this topic and delve into more advanced topics related to I/O in Tcl:

- Asynchronous input and output operations
- Transforming data during I/O
- Defining new channel types using Tcl's reflected channel abstraction

21.1. Asynchronous I/O

All the I/O operations we have seen so far with files as well as process pipelines have involved *blocking* I/O where the invoked command, `gets`, `read` etc., will not return until the I/O operation is completed or the channel end-of-file is reached. For channels backed by files, this behaviour is acceptable for the most part. For channels where the incoming data is intermittent and not always immediately available, this mode of operation is undesirable since the process may be blocked from doing any useful work for long intervals.

For example,

- The application may fire off a process pipeline to carry out a long computation. If it is blocked while reading from the pipeline, it cannot respond to the user or do any other tasks until the computation completes.
- A network server waiting for data from a client will be blocked from communicating with other clients and can effectively only service one at a time.
- Serial port communication is slow to begin with and if there is a human at a terminal on the other end, there is a lifetime between character arrivals. Unless the application is dedicated to responding to that device, it is not feasible to block while waiting for data to show up on the port.

These are the types of situations for which *non-blocking* I/O is designed. When a channel is in non-blocking mode, the command will return immediately even if the I/O operation cannot be completed. The application can then attempt to try the operation at a later point.

There is one issue that arises with non-blocking I/O and that is with regard to when the application retries the I/O operation. Polling continuously is no better than blocking and polling at intervals is neither efficient nor responsive. It would be nice if there were a mechanism whereby the application is notified when a channel is ready for the required

operation. As always, Tcl doesn't disappoint! Channels can deliver such notification events through the same machinery we described in Chapter 19.

Non-blocking channels and channel event notifications are almost always used in combination to perform *asynchronous I/O*. Channels are set to non-blocking mode and any I/O operations take place only in response to channel events.

For ease of exposition however, we will start off with a description of non-blocking operations without involving channel events.

21.1.1. Non-blocking I/O

21.1.1.1. Changing the blocking mode for a channel

```
chan configure CHAN -blocking BOOLEAN
```

A channel's blocking mode is controlled with the `-blocking` option to the `chan configure` command (Section 13.5). The channel *`CHAN`* is set to blocking mode if *`BOOLEAN`* is a boolean true value and to non-blocking otherwise. Note that setting the blocking mode affects both read and write operations. It is not possible to set them independently.

21.1.1.2. Checking if a channel is blocked

```
chan blocked CHAN
fblocked CHAN
```

The `chan blocked`, and the equivalent `fblocked`, command returns `1` if the last input operation on the channel failed because there was not enough data available to satisfy the request, and 0 otherwise. We will see examples where it is needed later in the chapter.

21.1.1.3. Non-blocking input

A read operation on a channel may block because both the channel and device buffers are empty or contain less data than requested. The effect of this condition on non-blocking read operations depends on the command invoking the operation.

21.1.1.3.1. Reading lines in non-blocking mode: chan gets, gets

```
chan gets CHANNEL ?VARNAME?
chan pending DIRECTION CHANNEL
gets CHANNEL ?VARNAME?
```

The `chan gets`, and equivalent `gets`, command retrieves a single line from a channel. In non-blocking mode, **if a complete line is available** (either because a newline is seen or EOF is reached after reading at least one character), the command behaves the same as in blocking mode as described in Section 13.7.1. In the case where **a complete line is not available**, non-blocking mode differs from blocking mode operation:

- If *`VARNAME`* is specified, the command returns `-1`. The variable is not affected.
- If *`VARNAME`* is not specified, the command returns an empty string as its result.

A complete line may not be available either because the channel is at end of file, or the channel is not at end of file but the received data does not (yet) contain a line terminator that would form a complete line.

The following commands can be used to distinguish the two cases:

- The `chan eof` command returns `1` if the channel is at end of file and `0` otherwise.
- The `chan blocked` command returns `1` if the channel is not at end of file but is blocked because the incoming data does not form a complete line. Otherwise it returns `0`.

When an attempted read fails because a complete line is not available, it can be useful to know how much data is buffered internally by the channel. That way, if it exceeds some maximum permitted line length, in a network protocol for instance, we can take appropriate action such as terminating the connection. The `chan pending` command will return the number of buffered bytes (**not** characters). The *`DIRECTION`* argument to `chan pending` may be `input` or `output` depending on whether we are interested in the input or output side of the channel. We'll see an example below.

OK, enough of the theory. Let us experiment with non-blocking behaviour using a pipe channel (Section 20.3). You will recall that data written to the output end of the pipe can be read from the input end. We create the pipe and assign the input and output channels.

```
lassign [chan pipe] in out → (empty)
```

We put the output side into line buffering mode so it will flush automatically any time it sees a newline character.

```
chan configure $out -buffering line → (empty)
```

We are now ready to communicate over the pipe channel. First we will put the input end into non-blocking mode and attempt to read a line. Since we have not written to the pipe as yet, the input buffer will be empty.

```
chan configure $in -blocking 0 → (empty)
chan gets $in line             → -1
```

The returned character count is `-1` as expected. Attempting a read using the second form of `gets` returns an empty string. We can confirm that the channel is not at end of file and that there is no line in the input buffer.

```
chan gets $in    → (empty)
chan eof $in     → 0
chan blocked $in → 1
```

Now let us write two **empty** lines to the pipe.

```
chan puts $out "" → (empty)
chan puts $out "" → (empty)
```

We use the first form of `gets` to attempt to read a line.

```
chan gets $in line → 0
```

The return value of `0` indicates that an empty line was read. Let us read the second empty line without passing the variable argument.

```
set line [gets $in] → (empty)
string length $line → 0
```

Again we get back an empty line. How do we know whether it is a "real" empty line, or an incomplete one, or end of file? Again, we use `chan eof` / `eof` and `chan blocked` / `fblocked` to find out.

```
chan eof $in     → 0
chan blocked $in → 0
```

Both return `0` so we know it was indeed a empty line sent by the "remote" end.

Let us then write data to the channel without an end of line character. Notice we need to do an explicit flush to make sure the data is sent to read side of the channel.

```
chan puts -nonewline $out "A few words" → (empty)
chan flush $out                         → (empty)
chan gets $in line                      → -1
chan gets $in                           → (empty)
chan eof $in                            → 0
chan blocked $in                        → 1
```

Even though there is data in the input buffer, the two forms of the `chan gets` command returned `-1` and an empty string respectively, as the buffered data did not form a complete line. The `chan eof` and `chan blocked` calls confirmed as much.

Do not confuse the buffering mode we set with `-buffering line` with line completion handling. The `-buffering` option only controls when data is flushed from the output buffers.

We can in fact confirm there is data in the input buffer with `chan pending`.

```
chan pending input $in → 11
```

The command tells us there are 11 bytes pending in the input.

Let us now examine the end of file condition. We close the output side of the pipe and attempt another read.

```
chan close $out → (empty)
chan gets $in   → A few words
```

Notice that on end of file, the buffered input content is delivered even though no newline characters were present.

Subsequent reads then indicate end of file which we can check with `eof`.

```
chan gets $in line → -1
chan gets $in      → (empty)
chan eof $in       → 1
chan blocked $in   → 0
chan close $in     → (empty)
```

21.1.1.3.2. Reading characters in non-blocking mode: chan read, read

```
chan read ?-nonewline? CHANNEL
read ?-nonewline? CHANNEL
```

We described the basic operation of the `chan read`, and the equivalent `read`, commands in Section 13.7.2. The commands have two forms, one where the number of characters to read is not specified.

In the other, the caller explicitly asks for a specific number of characters.

```
chan read CHANNEL NUMCHARS
read CHANNEL NUMCHARS
```

In blocking mode, the commands read a specified number of characters from a channel or all characters till end of file if no character count is specified. If the specified number of characters is not available, or end of file is not reached in the case of no count being specified, the `read` command will wait.

In non-blocking mode, both forms alter their behaviour to return whatever data is available even if less than requested. We will again use a pipe for demonstration. This time since we are writing a character stream, we will set the buffering mode to `none` to ensure the data is passed along right away.

```
lassign [chan pipe] in out          → (empty)
chan configure $out -buffering none → (empty)
chan configure $in -blocking 0      → (empty)
```

An attempt to read at this point returns empty strings. As before we can use `eof` and friends to check the cause.

```
chan read $in    → (empty)
chan read $in 1  → (empty)
chan eof $in     → 0
chan blocked $in → 1
```

Let us write a single character and attempt to read two.

```
chan puts -nonewline $out A → (empty)
chan read $in 2             → A
```

As you can see, the read returns a single character which is all that was available in the input buffer. The same holds true if we tried to read to end of file.

```
chan puts -nonewline $out B → (empty)
chan read $in               → B
chan eof $in                → 0
```

When closing the pipe, we take the opportunity to reiterate another point about end of file conditions. When we check for end of file after the pipe is closed, `eof` returns `0`, **not** `1`.

```
chan close $out → (empty)
chan eof $in    → 0
```

This is because `chan eof` / `eof` **only detect an end of file after an input command (** `gets` **or** `read`**) fails** because of an end of file condition. To prove that point,

```
chan read $in  → (empty) ❶
chan eof $in   → 1
chan close $in → (empty)
```

❶ Empty string returned due to EOF

21.1.1.4. Non-blocking output: chan puts, puts

```
chan puts ?-nonewline? ?CHANNEL? DATA
puts ?-nonewline? ?CHANNEL? DATA
```

The non-blocking behaviour on the output side using `chan puts` or `puts` is a lot simpler than on the input side.

The basic operation of these commands in blocking mode was covered in (Section 13.6).

In non-blocking mode, Tcl will accept the data and store it in its internal buffer, growing it as necessary. When the device is ready to accept more data, this internal buffer is flushed to the device behind the scenes. However, **this requires that the Tcl event loop be running**. This is not generally a problem because non-blocking I/O is almost always used in conjunction with channel events to implement asynchronous I/O.

In non-blocking mode, even the `flush` / `chan flush` commands do not (can not) flush the internal buffers to the device if it is not ready. They will be flushed in the background, potentially after the command returns. To force data to be written out immediately, the channel must be placed in blocking mode, flushed and then reverted to non-blocking mode. Of course, this blocks the application until the flush completes.

Although Tcl will accept any amount of data on output in non-blocking mode, it is advisable to use channel event notifications to only write when the channel is ready to accept data. Otherwise, there is a danger of the output buffers growing unacceptably large causing memory pressure.

And that leads us to a discussion of event-driven I/O.

21.1.2. Event driven I/O: chan event, fileevent

```
chan event CHANNEL EVENT ?HANDLER?
fileevent CHANNEL EVENT ?HANDLER?
```

As we discussed earlier, efficient asynchronous I/O requires some means for an application to be notified when a channel is ready to receive data on output or has data available for reading. The channel subsystem in Tcl provides these notifications by generating events when channels are ready for input or output. An application can then register callbacks to be invoked on the occurence of these events.

Channel events are tied into the same eventing infrastructure we described in Chapter 19 and require the event loop to be running. On every iteration of the event loop, each channel driver is given an opportunity to add events to the event queue. If any channel event handlers are registered for a channel and it has notifications pending, the channel driver will enqueue an entry to event queue with the event details and handler information. When the event loop processes the event queue, it will invoke any queued handlers.

The handlers for channel events are registered with the `chan event`, or equivalent `fileevent`, command. *`HANDLER`* is the callback script that should be invoked in reaction to the notification event *`EVENT`*. If a handler is already registered for *`EVENT`* on the channel, it is replaced. If *`HANDLER`* is the empty string, the currently registered handler, if any, is unregistered. If the *`HANDLER`* argument is not specified, the command returns the script currently registered for that channel for the *`EVENT`* event or an empty string if none is registered.

`EVENT` must be one of `readable` or `writable`.

A `readable` event is generated under either of two conditions:

- Data is available to be read from the channel.
- The channel is at end of file.

As we will see in our examples, the handler script generally uses the methods described in the previous sections to read data or detect end of file on the channel.

A `writeable` event is generated when the channel is ready to accept data from the application. A special case of this condition occurs on asynchronous network socket connections when the connection set up is completed. We will look at sockets in Chapter 22.

As a first example of event driven I/O, let us revisit our pipe examples from the previous section except that instead of blindly reading from the pipe, we will only attempt to read when we are notified that data is available. The example also highlights some points you need to be aware of regarding asynchronous I/O so we will go through it slowly.

```
lassign [chan pipe] in out                      → (empty)
chan configure $out -buffering none -blocking 0 → (empty)
chan configure $in -blocking 0                  → (empty)
```

We have turned off buffering for the output channel so that data is immediately written to the pipe for reasons we will see later.

Next, we write a procedure that implements the event handler. This will be invoked for every `readable` event on the channel of interest.

```
proc read_handler {chan} {
    set status [catch {gets $chan line} nchars]
    if {$status == 0 && $nchars >= 0} {
        puts "Received: $line"
        return
    }
    if {$status || [chan eof $chan]} {
        puts "All done!"
        chan event $chan readable {}
        set ::exit_flag 1
        return
    }
    puts "Incomplete line"
}
```

The handler starts off with attempting to read a line from the channel. We use the `catch` command to trap any possible errors that the `gets` command might raise, for example, if the remote end aborts a network connection. If there were no errors and the command read a line successfully (`nchars` is not negative), we print the line and return.

Otherwise, if either an error occurred on the channel read or if the channel is at end of file, we remove the read handler from the channel, set an exit flag and return. This global `exit_flag` is needed because we will be using `vwait` to run the event loop for our little example. In a real application already running the event loop, this would not be necessary.

When none of the above conditions are met, the procedure falls through to print the `Incomplete line` message. We will see in a bit the conditions under which this can happen.

When an end of file is seen on a channel, it is crucial to either remove the read handler from the channel as we have done, or to close the channel in the handler itself before returning. Otherwise, the channel will continuously raise `readable` events because the channel is at end of file.

Now we register our read handler for the input channel. We have to explicitly pass in the channel as an argument to the handler since the channel subsystem does not itself pass in this information. Just for kicks, we also confirm that it is registered.

```
chan event $in readable [list read_handler $in] → (empty)
chan event $in readable                         → read_handler file24154e677c0
```

Having set up the read handlers on the input channel, it is time to write to the pipe and see what happens. To simulate data arriving intermittently as from a remote network, we will split up the writes into partial writes with intervening delays, all scheduled through the event loop with `after` (Section 19.3.2). This is also the reason why we turned off buffering for the output channel earlier.

```
after 50 [list puts $out "Hello World!"]         → after#41
after 100 [list puts -nonewline $out "Goodbye "] → after#42 ❶
after 150 [list puts $out "World!"]              → after#43
after 200 [list close $out]                      → after#44
```

❶ Note incomplete line

Finally, we get the event loop rolling with `vwait` (Section 19.2.3.1). In a real application that expected to do asynchronous I/O, the event loop would already be running but that is not true for our interactive shell. So we have to explicitly start it.

```
% vwait ::exit_flag
→ Received: Hello World!
  Incomplete line
  Received: Goodbye World!
  All done!
% close $in
```

Let us now examine the resulting output.

- When the first string written to the channel is received on the input side, our read handler is invoked as there is data available on the channel. Since an entire line is available, the `gets` call succeeds and the line is printed out.
- The second string written to the channel is an incomplete line. When the data arrives on the input side, our read handler is again invoked. This time the `gets` returns `-1` because a complete line is not available in the input buffer. The first `if` condition fails. Moreover, since the end of file is not reached, the second `if` condition also fails, and control passed to the bottom of the procedure resulting in `Incomplete line` being printed. As a quirk of the Tcl channel implementation, the arrival of data may result in additional events being posted resulting in the line being printed more than once.
- The third write results in another invocation of the read handler with a complete line being available and printed.
- Finally the sending side closes its end of the pipe. The resulting end of file on the input side also triggers the read handler. This time the `gets` returns `-1` and `eof` indicates end of file. Consequently, the second `if` block is executed. **Here we are careful to remove the read handler from the channel otherwise we would be continuously invoked.** We could have closed the channel instead but we leave that for the main line code.
- Because the read handler set the `exit_flag`, the `vwait` command terminates the event loop and returns. We go on to close the input channel.

In our example, we used `gets` to read the channel a line at a time. We could also have used the `chan read` (Section 13.7.2) / `read` (Section 13.7.2) commands as well keeping in mind the differences with respect to `gets` that we described in Section 21.1.1.

The above example dealt with read handlers. We will not show an example of a write handler here as we will discuss further examples in Chapter 22.

One final note about channel event handlers. If the event handler raises an uncaught exception, Tcl will unregister it from the channel.

21.1.3. Closing non-blocking channels

There are a couple of considerations with regards to closing non-blocking channels.

- The `close` command on a non-blocking channel returns immediately before any buffered data is written out. The data is then flushed in the background. This requires the event loop to be active. Moreover, the application should not assume the data is will be available on the underlying device when the `close` command returns.
- Any open non-blocking channels must be switched to blocking mode before exiting the process. Otherwise any buffered data may not be written out before the process exits.

21.1.4. An interactive command line

When you run `tclsh` without any arguments, it enters an interactive read-eval-print-loop (REPL) where it executes Tcl scripts entered on the command line. This default REPL uses blocking I/O and does not have the event loop active. In order to have any event loop based functionality, such as asynchronous I/O, you need to enter the event loop through a command such as `vwait`. However, you then lose the ability to enter commands at the command line.

Another common situation that arises is in an event loop based application such as a network server. The script implementing the server usually has a `vwait` at the end that activates the event loop. Although these are generally "background" applications, it is nevertheless useful for them to be able to expose an interactive command line interface for purposes such as configuration, troubleshooting etc.

An **event-driven REPL** is useful in both the above scenarios. It collects interactive command input using non-blocking I/O permitting other event processing to proceed without interruption. A basic implementation is shown below.

```
namespace eval repl {}
proc repl::prompt {prompt} {
    puts -nonewline stderr $prompt
    flush stderr
}
proc repl::repl {} {
    variable command ""
    variable done
    prompt "% "
    fileevent stdin readable [namespace current]::repl_handler
    vwait [namespace current]::done
}
```

The `repl::repl` procedure is intended to be called from the application to enter the event loop while displaying a REPL for command input. It does some initialization, sets up a read handler on standard input and then enters the event loop.

The read handler `repl_handler`, shown below, is where the hard work is all done. It is invoked when a full line is available on the standard input or if the input channel is closed. In the latter case, it simply terminates the `vwait` loop after removing itself as the input handler. Otherwise, it appends the new line to any previously collected input. If the command is syntactically complete, it is executed in the global scope and the result printed. If the command is not complete, the secondary prompt is displayed.

```
proc repl::repl_handler {} {
    variable command
    if {[gets stdin line] < 0} {
        fileevent stdin readable {}
        set [namespace current]::done 1
        return
    }
    append command $line
    if {[info complete $command]} {
        fileevent stdin readable {} ❶
        set status [catch {uplevel #0 $command} result]
        fileevent stdin readable [namespace current]::repl_handler
        if {$result ne ""} {
            if {$status == 0} {
                puts $result
            } else {
                puts stderr $result
            }
        }
        set command ""
        prompt "% " ❷
    } else {
        append command \n
        prompt "> " ❸
    }
}
```

❶ Avoid nested calls
❷ Primary command prompt
❸ Secondary command prompt

One point to be noted above is that the read handler disables itself before calling `uplevel` and then restores itself afterward. This is a precautionary measure in case the script executed in the `uplevel` call itself recursively enters the event loop. In that case we do not want our read handler to be called if more input is available until the currently executing `uplevel` has finished execution.

The above implementation leaves out some details for pedagogic purposes. More complete implentations are available in the Tcler's Wiki, for example see https://wiki.tcl-lang.org/page/commandloop.

21.2. Channel transforms

In Section 13.12 we described the use of the `-encoding` option to transparently encode data written to a channel. This removes the burden from applications to remember to explicitly encode strings when writing to a file. This is very convenient when the output commands may be spread over multiple locations in the application.

Now consider an application writing to a log file where the data must be compressed to save disk space. Or perhaps encrypted in some form to hide passwords or other private details. Clearly transparency similar to that afforded by the `-encoding` option would be very convenient here as well.

Obviously Tcl cannot have built-in facilities for all the infinite ways that data might be transformed. So it provides something almost as good — a way for applications to implement their own *channel transforms* that can alter the data stream flowing through a channel. Moreover, multiple channel transforms can be applied to a channel in a stacked fashion so the output of one transform is fed into the next. Thus we can write a compression transform and an encryption transform to meet our needs and even combine them.

21.2.1. Channel transform basic operation

Figure 21.1 shows data flow in a channel when the command `puts` is invoked with encoding set to UTF-8 and line endings to CRLF.

The flow on the left side has no channel transforms applied. The Tcl channel top layer then encodes the input data, translating newlines to CR-LF pairs and emits a byte stream to the underlying device which may be a file, a network socket etc.

The right side of the figure shows the flow with two transforms pushed onto the channel, first the `base64` transform that we assume converts binary data into base64 encoding and a second transform which compresses[1] the data using the DEFLATE algorithm we discussed in Section 8.5. This is not necessarily a sensible combination of transforms but ... whatever. It serves our illustrative purpose.

As shown in the diagram,

- The top layer of the Tcl channel subsystem produces a byte stream (or a binary string in Tcl parlance) and this is what the channel transforms see. There are no “characters” at this level so if the transform wants to do character based operations like upper-casing all text, it gets trickier than you might think. We will revisit this issue later.
- When multiple transforms are pushed on to a channel, the later ones are placed on top of earlier ones. Hence the term “stacked” transforms.
- The channel subsystem calls each transform in turn, passing it the data returned by the previous transform and using the data returned by the transform as the input to the next. The data produced by the bottommost transform is written to the device.
- Although not depicted in the figure, a transform is free to return an empty value in which case the transform below is not called. An example of this behaviour would be exhibited by block oriented encryption algorithms which operate on fixed length sequences of bytes. In

[1] Short strings like in our example will actually land up being longer

general, a channel transform may convert and pass on all, some or none of the data passed to it. The rest can be buffered internally for later processing.

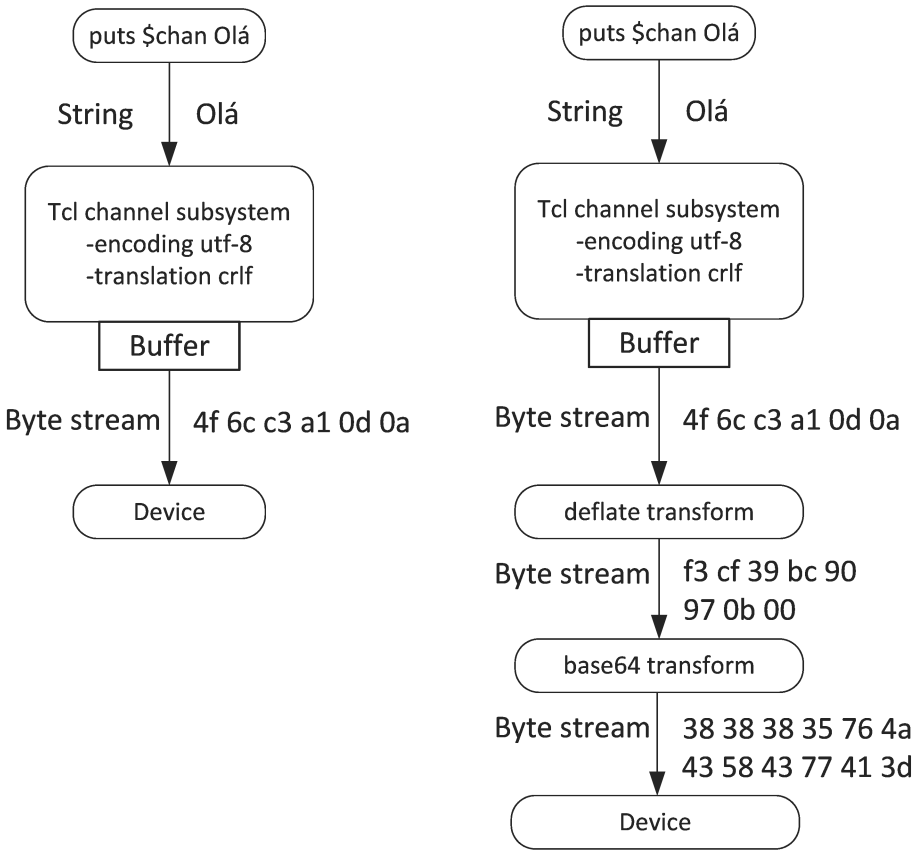

Figure 21.1. Basic channel operation

The above discussion described operations on the channel output path. For the input path, the operations are very similar, except in reverse order, and we will not describe them separately.

21.2.2. Implementing channel transforms

The Tcl channel subsystem expects channel transforms to be implemented in the form of a command prefix that will be invoked with arguments such as `read` that are subcommands indicating the operation to carry out. A channel transform may be implemented as namespace ensemble, a TclOO object or even as a simple procedure whose first argument specifies the operation.

The complete list of subcommands to be implemented by channel transforms is shown in Table 21.1. Note some are required for input transforms, some for output and some applicable to both.

Table 21.1. Channel transform subcommands

Subcommand	Direction	Purpose
`initialize` *`HANDLE MODE`*	Both	Initialize the transform.
`finalize` *`HANDLE`*	Both	Finalize and clean up resources.
`clear` *`HANDLE`*	Both, optional	Clear internal buffers or state.
`drain` *`HANDLE`*	Read, optional	Return any internally buffered data to the channel subsystem so it can be passed to the higher layers.
`limit?` *`HANDLE`*	Read, optional	Limit the number of bytes read ahead by the Tcl channel subsystem.
`read` *`HANDLE BUFFER`*	Read	Process data read from the device or transformer below this transform.
`flush` *`HANDLE`*	Write, optional	Pass internally buffered data to the underlying device or transform.
`write` *`HANDLE BUFFER`*	Write	Process data from the layer above. The returned result, if any, will be passed to the layer below.

Note that a channel transform need not be applicable to both input and output sides of a channel. For input-only and output-only transforms, the `write` and `read` subcommands respectively, need not be implemented.

Let us examine the implementation of these subcommands in detail by working through an example. We will implement a channel transformation for encrypting data using the RC4 cipher.

The use of RC4 in practice is not recommended because of vulnerabilities in the algorithm. We use it here because of the simplicity of the RC4 interface allows us to focus on the channel abstraction itself.

The RC4 implementation we use is from the `rc4` package of Tcllib[2]. We start by defining a TclOO class that implements the transform wherein the subcommands shown in Table 21.1 are methods of the class. For ease of exposition, we will construct the class piecemeal.

```
package require rc4
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
}
```

[2] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

Since encryption requires keying material, we pass it as an argument to the constructor. The constructor passes this to the RC4 package to get back handles that can be used for encryption operations on output and decryption on input. We store these away into member variables.

21.2.2.1. Initializing channel transforms

Next we define the mandated `initialize` subcommand.

```
oo::define RC4Transform {
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
}
```

The first parameter of all subcommands of a channel transformation is a handle to the channel transform (**not** the owning channel). Implementations based on namespaces or procedure calls generally make use of this handle, called `transform_handle` in our sample code, to distinguish between different transform instances applied to different channels. With a TclOO based implementation like ours, the context is implicit in the object instance because a different instance is pushed onto each channel. Thus this handle is not of much use to our implementation and it is ignored in all methods.

The second parameter to `initialize` is a list of one or two elements, from the strings `read` and `write`, indicating whether the attached channel is readable and writable. Our transformation does not care so we will ignore this parameter as well.

The return value from `initialize` should be the list of subcommands supported by the transformation.

21.2.2.2. Finalizing channel transforms

The other mandated command is `finalize` which is called by the Tcl I/O system when the transformation is popped from the channel.

```
oo::define RC4Transform {
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
}
```

The transformation handle is the only parameter passed and we ignore it for the reasons explained above. The purpose of the call is to allow the transformation to release any resources. In our case, we release the resources associated with the RC4 handles. No further calls are made to the transformation by the I/O system once this call returns.

We also choose to have the transformation commit suicide by calling its `destroy` method. By doing so, we free the application from having to keep track of the transformation object itself and destroying it at some suitable point. We could have also chosen to leave it to the application to destroy the object after the channel transformation is popped from the channel.

This is useful when the channel transformation may hold some additional computed value, such as a checksum, based on the data passing through the channel.

21.2.2.3. Transforming data

Finally, we come to the actual data transfer and implementation of the `read` and `write` subcommands.

```
oo::define RC4Transform {
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
```

The `read` and `write` commands both take two argments, the channel transformation handle which we ignore as before, and the binary data to be transformed. Our implementation is very simple as we leave it up to the RC4 package to do the hard work.

Here then is the complete channel transformation in a condensed form.

```
package require rc4
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read  [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
→ ::RC4Transform
```

We will look at how we could use our RC4 transform in a bit but first we take a short detour to discuss some buffering issues that are not relevant to our example because of its simplicity but need to be managed in more complex transforms.

21.2.2.4. Buffering in channel transforms

Our example was greatly simplified by the fact that we had no need for internal buffering. In particular, data coming in to the transform in either direction was immediately passed on.

We now look at issues that arise when this is not possible due to the characteristics of the transform. Examples include

- Transforms that are block oriented such as the base64 encoding scheme or the DES encryption algorithm. These require the data to be transformed in multiples of fixed size blocks. Since there is in general no control over how the Tcl I/O system and other transforms pass in data, partial blocks have to be buffered internally and then transformed when further calls complete the block.
- Some transforms such as `zlib` compression are stream oriented but perform better when the data is compressed in larger chunks. These may choose to internally buffer data for performance reasons.
- Some transforms may deal with variable length blocks. These are the most complex to deal with. A common example of this kind of transform is anything dealing with characters as opposed to bytes. Consider a transform that changes the character case of all data passing through to upper case. Depending on the character encoding in effect on the channel, each character may be encoded as a variable number of bytes. The transform then has to parse the byte stream to recover and transform individual characters. To further complicate matters, a multibyte character may be split across multiple `read` (or `write`) calls.

At some point, such as when the channel is closed or the channel transform is popped from the channel, Tcl has to ask the transformation to flush its internal buffers and return their content. It does this by calling the `flush` and `drain` subcommands. The former is called for the output side of the channel and the latter for the input. Both commands take the transformation handle as their only parameter and are expected to complete transforming the data in their internal buffers and return the result.

The other situation concerning internal buffers involves the application calling `chan seek` (Section 13.15.2) on the channel. Since this changes the access pointer for the file (or any channel that supports seeks), internal buffers must be cleared. In this situation, the Tcl I/O subsystem calls the `clear` subcommand. In response, the channel transformation is expected to clear its internal buffers, **both** input and output, throwing away all stored data in the process.

Channel transformations that do internal buffering generally need to implement these three subcommands.

Applications should avoid calling `chan seek` or `seek` on channels which have transforms applied. By their very nature, transforms like compression are not amenable to these operations.

We will not present any examples of channel transformations that do internal buffering. You can look at the source code for the `tcl::transform::base64` package in Tcllib[3] for a real world example.

[3] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

21.2.2.5. Limiting read-ahead

For reasons of performance, in normal operation the Tcl I/O system will read more bytes from a device than are actually requested by the application. This can be a problem for transforms that expect data streams in some bounded format. Consider using a transform on network channel to decompress a compressed chunk of an HTTP stream. If the Tcl I/O system reads ahead and passes data beyond the compressed portion, the transform will process the compressed part but has no way of returning the remaining portion back to the I/O system if the transform is then removed from the channel. (The application in this case knows how many bytes are compressed and pops the transform after reading that number.)

The channel transformation can implement the `limit?` command to control the amount of read-ahead done by Tcl on input. It takes the channel transformation handle as argument and should return the upper limit for the number of bytes that Tcl is allowed to read ahead, returning a negative integer to indicate no upper limit. It is called by Tcl before every `read` subcommand invocation. The subsequent `read` subcommand invocation will never pass in data of length more than the (non-negative) value returned by the `limit?` call.

21.2.3. Using channel transforms: chan push|pop

```
chan push CHANNEL CMDPREFIX
chan pop CHANNEL
```

Let us now look at how an application utilizes a channel transform using `chan push` and `chan pop`.

The `chan push` command places a channel transform at the top of the channel transform stack, i.e. just below the Tcl I/O buffering layer as shown in Figure 21.1. Successive `chan push` commands result in a stack of transformations with the last one pushed on top.

The `chan pop` command removes the topmost channel transform from the specified channel. It is not possible to remove any transform other than the topmost.

The *`CMDPREFIX`* is a command prefix callback that in effect identifies the channel transform. When carrying out various I/O operations, Tcl invokes *`CMDPREFIX`* with one of the subcommands from Table 21.1.

We will use our `RC4Transform` channel transformation across a pipe for demonstrative purposes. First we create an instance of the transformation and the pipe.

```
set xform [RC4Transform new secret] → ::oo::Obj1208 ❶
lassign [chan pipe] in out          → (empty)
```

❶ It is a terrible idea to use an ASCII string for keying but this is just a demo

We will apply this transform only on the **output** side of a pipe. On the other end of the pipe, we will simply read the channel in binary mode without the transform applied. In effect, this will tell us what bytes are being written to the underlying operating system pipe.

```
chan push $out $xform                  → file24154b7fbd0
chan configure $in -translation binary → (empty)
puts $out "Just a demo"                → (empty)
close $out                             → (empty)
```

Reading the other end of the pipe tells us the raw bytes written from the output side. We will dump the data in hex using our `bin2hex` helper procedure.

```
bin2hex [set encrypted [read $in]] → a7 43 a1 68 a2 c5 f6 c2 57 a6 d4 d1 f9
close $in                          → (empty)
```

As you can see the data was encrypted, something we can verify by decrypting with the `rc4::rc4` command.

```
rc4::rc4 -key "secret" $encrypted → Just a demo
```

Of course, in a real application both ends of the communication would have a transform applied. Obviously, the same shared secret has to be used for both.

```
set in_xform [RC4Transform new secret]  → ::oo::Obj1209
set out_xform [RC4Transform new secret] → ::oo::Obj1210
lassign [chan pipe] in out              → (empty)
chan push $out $out_xform               → file24154afabd0
chan push $in $in_xform                 → file241549688b0
puts $out "Just a demo"                 → (empty)
close $out                              → (empty)
read $in                                → Just a demo
close $in                               → (empty)
```

21.2.4. Zlib channel transforms

```
zlib push ALGORITHM CHANNEL ?OPTIONS?
```

In Section 8.5, we introduced the `zlib` command for compressing binary strings. In addition to the features described there, the command also directly supports use as a channel transform with the `zlib push` command. which pushes a zlib transformation on to the specified channel. This transform may be popped with the normal `chan pop` command.

The *`ALGORITHM`* argument specifies the type of compression or decompression to be applied and must be one of `deflate` or `inflate` (Section 8.5.1.1), `compress` or `decompress` (Section 8.5.1.2), `gzip` or `gunzip` (Section 8.5.1.3).

The command takes the options shown in Table 21.2.

Table 21.2. zlib push command options

Option	Description
`-dictionary` *`BINDATA`*	See Table 8.6.
`-header` *`HEADER`*	See Table 8.6.
`-level` *`LEVEL`*	See Table 8.6.
`-limit` *`LIMIT`*	Specifies a limit for read-ahead of data. See Section 21.2.2.5. Defaults to `1`.

Once the transform is pushed on a channel, the `-dictionary`, `-header` and `-limit` options can be read with the `chan configure` command.

In addition, two additional options can be passed to `chan configure`. The `-checksum` option is read-only and returns the checksum for the uncompressed data seen on the channel up to that point. The write-only `-flush` option takes values `sync` or `full`. See Section 8.5.2.9 for their effect.

A simple example of using the transform follows. We create a channel to a file, push the compression transform onto the channel, and write to it.

```
set chan [file tempfile path]              → file24154a77010
zlib push compress $chan                   → file24154a77010
puts $chan [string repeat "abcdefghij" 10] → (empty)
```

Let us also retrieve the checksum before we close the channel.

```
flush $chan                    → (empty)
chan configure $chan -checksum → 525608894
close $chan                    → (empty)
```

Notice that we flush the channel before retrieving the checksum. Otherwise the transformation would not have seen all the data yet. If the buffering on the channel is turned off, this is not necessary.

Let us see if the file is compressed.

```
file size $path → 23
```

Indeed it is. But let us read it back to ensure it is compressed and not truncated. Along the way, we retrieve the checksum and see that it is correct.

```
set chan [open $path r]        → file24154c6fd30
zlib push decompress $chan     → file24154c6fd30
read $chan                     →
 abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefg...
chan configure $chan -checksum → 525608894
close $chan                    → (empty)
```

One point to note is that even though the compressed file is a binary file, not text, we do not open it in binary mode. The mode for `open` is the same mode used to write the file (defaulting to text in our example). The transformation sits **below** the text / binary translation layer of the Tcl I/O subsystem so it will always see the raw bytes from the file irrespective of how the channel is configured.

21.3. Reflected channels

Whereas channel transforms allow us to process data as it is being passed through channels, Tcl's reflected channel facility presents a way to define entire new channel types at the script level. With reference to Figure 21.1, reflected channels occupy the "device" box in the I/O stack.

Our discussion is mostly focused towards **implementation** of reflected channels as **using** them is for the most part no different from the built-in channels. However, there are some notable limitations that we describe in Section 21.3.3.

21.3.1. Implementing reflected channels

Just like channel transforms, reflected channels are implemented as command prefixes that are invoked by Tcl's I/O system in response to various I/O operations. These command prefixes may be in the form of namespace ensembles, TclOO objects or anything else that will accept subcommands that specify the operation.

The complete list of these subcommands is shown in Table 21.3. As for channel transforms, implementation of some of these subcommands is optional depending on whether the channel supports read or write operation.

Table 21.3. Reflected channel subcommands

Subcommand	Direction	Purpose
`initialize HANDLE MODE`	Both	Initialize the channel.
`finalize HANDLE`	Both	Finalize and clean up resources.
`watch HANDLE EVENTS`	Both	Register the types of I/O events, read and/or write, that should be reported.
`read HANDLE COUNT`	Read	Return `COUNT` bytes from the channel.
`write HANDLE BYTES`	Write	Write the `BYTES` binary string.
`seek HANDLE OFFSET BASE`	Both, optional	Move the channel access pointer.
`configure HANDLE OPTNAME OPTVAL`	Both, optional	Set a channel configuration option.
`cget HANDLE OPTNAME`	Both, optional	Get a channel configuration option.
`cgetall HANDLE`	Both, optional	Get all channel configuration options.
`blocking HANDLE MODE`	Both, optional	Set the blocking mode of a channel

As before, we will look at these subcommands in detail as we go through the implementation of a simple reflected channel. Our illustrative sample implements a channel that will simply store data written to it and allow it to be read out. The `tcl::chan::fifo` module in Tcllib[4] already provides such a reflected channel but ours will differ in a couple of respects for pedagogic purposes.

Whereas we demonstrated channel transformations using TclOO, for reflected channels our implementation will use namespaces instead so they don't feel left out. All our subcommands will be implemented as procedures within this namespace which we will name `bureaucrat`. Why, you ask? Because

- It introduces unnecessary delays in data transfer without doing any useful work in the process. In our case, this is to better demonstrate event-driven I/O.
- It will sit on one task and refuse to accept another until the first one is done. This is for demonstrating flow control and how blocking is signaled to the Tcl I/O system.

For reasons that will become clear later, our reflected channel requires the event loop to be running.

21.3.1.1. Initializing a reflected channel

On channel creation, the Tcl I/O system will call the `initialize` subcommand for the reflected channel.

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}
}
proc ::bureaucrat::initialize {chan mode} {
    variable channels
    set channels($chan) {
        State    OPEN
        Data     {}
        InFlight 0
        Delay    100
        Blocking 1
        Watch    {}
    }
    return {
        initialize finalize
        read write watch blocking
        configure cget cgetall
    }
}
```

The first parameter of `initialize` is the channel name. Unlike a TclOO based implementation like the one we demonstrated for channel transforms, a namespace based one does not maintain an implicit per-channel context. We therefore store the per-channel state as a dictionary in an array `channels` indexed by the channel name. The semantics of the keys in the dictionary will be clarified as we go along.

[4] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

The second parameter to `initialize` is a list of one or two elements `read` and/or `write` indicating whether the attached channel is readable and writable. We ignore this parameter as we do not care.

The return value from `initialize` is the list of supported subcommands. We do not support the seek operation and therefore it is not included in the returned list.

21.3.1.2. Closing a reflected channel

When an application closes a reflected channel, Tcl invokes the `finalize` subcommand to allow it to clean up any allocated resources. Our toy reflected channel is self contained so all we need to do is get rid of the state variable.

```
proc ::bureaucrat::finalize {chan} {
    variable channels
    unset -nocomplain channels($chan)
}
```

21.3.1.3. Configuring a reflected channel

A reflected channel may **optionally** define its own configuration options in addition to the standard ones defined by Tcl for all channels. This requires the channel to implement the `configure` subcommand which is called when an application wants to set an option.

In our example, we will allow the channel delay to be configured by an application. The channel implements two subcommands `cget` and `cgetall` for retrieving configuration values. Again, these are optional. However, if either one is provided, the other one must be provided as well. The `cget` command should return the value of the specified option. The `cgetall` command should return a dictionary of all options and their values.

```
proc ::bureaucrat::configure {chan optname optval} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    dict set channels($chan) Delay $optval
    return
}
proc ::bureaucrat::cget {chan optname} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    return [dict get $channels $chan Delay]
}
proc ::bureaucrat::cgetall {chan} {
    variable channels
    return [list -delay [cget $chan -delay]]
}
```

The standard configuration options on channels are handled by the Tcl I/O system itself and never passed down to the reflected channel.

21.3.1.4. Non-blocking mode and event driven I/O

The next two subcommands we discuss, `blocking` and `watch` pertain to non-blocking and event-driven I/O operations.

Channels always start out in blocking mode. When an application sets or resets the blocking mode of a channel with a call of the form

```
chan configure $chan -blocking 0
```

the Tcl I/O system makes a call down to the channel's `blocking` subcommand to inform it of the expected blocking mode of operation. In our case, we simply save the mode in the channel's state. We will make use of its setting when we implement the actual reads and writes.

```
proc ::bureaucrat::blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}
```

While the `blocking` command is optional, the `watch` command is mandatory. It is called by the Tcl I/O system in response to the application indicating an interest in read and/or write events on the channel (Section 21.1.2).

```
proc ::bureaucrat::watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}
```

The `events` parameter to `watch` is a list, possibly empty, containing some combination of the values `read` and `write` indicating the channel events for which the application has registered an interest. The `Watch` element of our state dictionary keeps track of this information. We check if we are **adding** either event type as an event of interest. If so, the `notify` procedure, that we define next, is called to send an appropriate event to the application. We do this check to avoid sending extraneous events in case we were already notifying that event type.

The `notify` procedure is internal to our implementation and not called directly by the Tcl I/O system. Its purpose in life is to generate an event notification to the application if the conditions warrant it.

```
proc ::bureaucrat::notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}
```

The `notify` procedure is short but there are several important points to be noted therein so we will go over it line by line.

- First, if the application has indicated no interest in the event we return.
- Next we have to check if the conditions implied by the event are present.
- For `read` events, the channel must have data available to read **or** it must be at end-of-file.
- For `write` events, the channel must be in an open state.
- If these conditions are met, an event is generated with the `chan postevent` command.

We will discuss this last point in a bit but first we need to touch on the channel state conditions. In our simple example, the channel is always in only one state — that we designate as `OPEN` — from the time it is created to the time it is closed. Channels in the real world tend to be more complex. A network connection may transition from a "connecting" state to an "open" state, to an "end of file" state if the remote end closes the connection. Our `watch` procedure assumes such state transitions for pedagogic purposes even though the channel itself does not go through these states.

The `chan postevent` command is used to send events to the application.

```
chan postevent CHANNEL EVENTLIST
```

Here *`CHANNEL`* is the channel for which the event is to be generated and *`EVENTLIST`* is a list containing one or both of `read` or `write` denoting the events. In response, the Tcl I/O system will invoke any event handlers that were registered for the channel.

One final explanation is warranted for the manner in which `chan postevent` is called. Basically, `chan postevent` is scheduled through the event loop and not called directly. The reason for this is that `chan postevent` implementation will directly invoke the application event handler. That may in turn call back into any of the channel I/O API's and thereby back into our implementation even before we have completed the `notify` procedure. This reentrancy can get very tricky to deal with and we avoid it by scheduling through the event loop. That ensures that the current call is unwound completely before any application code calls into our code.

As to the use of the `after idle after 0` idiom, it prevents starvation of the event queue. See Section 19.3.3.1 for a full explanation.

21.3.1.5. Implementing data output

We finally come to implementing the main functions of a reflected channel, viz. some form of data transfer. We will begin with the output side of data transfer.

The Tcl I/O system will call the `write` subcommand to write data to the device. Although not directly relevant to our implementation, remember from our discussion of channel transforms that this is binary data after channel encoding and translation have taken place.

Our implementation of `write` has a couple of artifacts for the purposes of illustrating non-blocking and event driven operation. First, it introduces an artificial delay in the data transfer so that data is not immediately available on the input side. Second, it restricts data flow such that while the data is "on the way" to the input side, additional data will not be accepted.

```
proc ::bureaucrat::write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
        return 0
    }
    dict with channels($chan) {
        if {$InFlight} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
        }
        set InFlight 1
        after $Delay [list [namespace current]::delayed_receive $chan $bytes]
    }
    return [string length $bytes]
}
```

We implement the data transfer delay by scheduling a callback to the `delayed_receive` procedure with the `after` (Section 19.3) command. The `InFlight` element in the channel's state dictionary is used as a flag to remember that data is "in flight" via the event loop. Before the actual scheduling of the callback, we check this flag and if set, we return the `EAGAIN` error code to the caller.

The Tcl I/O system treats an error code of `EAGAIN` in special fashion. It is taken as an indicator that the channel cannot accept more data at this time, because its output buffers are full, or the remote end has flow controlled the connection and so on. The `chan blocked` command would then return `1` to the application. Tcl itself will continue to retry sending any buffered data at a later point via the event loop.

The error code used for indicating the blocking condition **must** be the string `EAGAIN`. You cannot use the integer value of this POSIX error instead as the value differs from system to system.

You may noticed something strange in the bodies of the `if` clause that checks the `Blocking` variable. Both bodies are the same! Normally, when `Blocking` is `true` the channel implementation should block and not return until either more data is available or an end of file condition occurs. However, we have no means of blocking and therefore are forced to deal with the condition in the same way as the non-blocking case. For a full explanation, see Section 21.3.3. **In a reflected channel that was backed by a OS descriptor, the implementation should block on that descriptor when `Blocking` is true instead of returning a `EAGAIN` error.**

The return value from the `write` should be the number of bytes that the channel accepted. In our implementation, we accept the whole data and thus return its length. However, this need not be the case. A channel may choose to accept only part of the data for whatever reason and return just the length of the accepted part. Tcl will then hold on to the remaining data and include it in a future call.

We now come to the "second half" of the output implementation. The `delayed_receive` procedure can be thought of as the equivalent of the procedure that will receive data from the remote end of a network connection.

```
proc ::bureaucrat::delayed_receive {chan bytes} {
    variable channels
    if {! [info exists channels($chan)]} {
        return;
    }
    dict with channels($chan) {
        if {$State ne "OPEN"} {
            return
        }
        append Data $bytes
        set InFlight 0
    }
    notify $chan read
    notify $chan write
}
```

The first thing `delayed_receive` does is check that the channel has not been closed between the time the procedure was queued and the time it was invoked. It also checks if the channel is still in the `OPEN` state (as we said before, this is not really required for our simple implementation). If both conditions are met, it appends the new data to the incoming data buffer. The `InFlight` flag is reset since there is no data queued in transit. It then calls the `notify` procedure we defined earlier to generate a channel read event so any waiting event handlers will be invoked. Since we were limiting writes while the data was in transit, we also need to generate a write event in case any handlers were waiting on the output side as well.

The need to generate a `write` event is really a peculiarity of our example because data is being input on the same channel as it is output. In most real world cases, there would be no need to generate a `write` event when **receiving** data.

Before moving on to implementation of the input side, one final point to reiterate is that the data written with the `write` subcommand should be treated as a byte stream with no direct

correspondence with `puts` commands at the application level. Due to encoding, translation and buffering by Tcl, not only might the number of bytes differ, but a single `puts` may be split into multiple `write` invocations, multiple `puts` may be combined into one and so on.

21.3.1.6. Implementing data input

The `read` subcommand is invoked by the Tcl I/O system when it needs to fetch more data from the channel. Because of Tcl's buffering, there is not always a direct correspondence between `gets` or `read` command invocations at the application level and the `read` subcommand of the channel.

The `read` subcommand is passed the channel name and the maximum number of input bytes to be read. If fewer bytes are available than requested, it is permitted to return a smaller number **but not more**. Returning zero bytes will be treated as an end-of-file condition by Tcl. If there are no bytes currently available but the channel is **not** at EOF, the procedure must signal this by raising `EAGAIN` error just as for the output side as we described earlier.

Given the above, our `read` implementation should be mostly self-explanatory.

```
proc ::bureaucrat::read {chan count} {
    variable channels
    dict with channels($chan) {
        if {[string length $Data] == 0} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
            return -code error EAGAIN
        }
        set bytes [string range $Data 0 $count-1]
        set Data [string range $Data $count end]
    }
    notify $chan read
    return $bytes
}
```

We do call `notify` at the end so that if there is still more data pending, the application read event handlers will be notified appropriately. With regards to both bodies of the `if` statement checking the `Blocking` condition being identical, see the explanation in Section 21.3.1.5.

21.3.1.7. Reflected channel creation: chan create

```
chan create MODE CMDPREFIX
```

Just two things remain to be implemented for our reflected channel.

- We have defined a bunch of procedures to implement the various operations expected of a reflected channel. However, the Tcl I/O system expects to invoke a single command prefix to which it appends the subcommand and arguments.
- We need to provide a way for the application to create our new channel type.

For the first task, we can simply create an ensemble command (Section 16.6).

```
namespace eval ::bureaucrat {
    namespace export *
    namespace ensemble create
}
```

The Tcl channel subsystem can then call our procedures as

```
bureaucrat initialize ...
bureaucrat read ...
```

and so on. Note our ensemble also makes visible our ancillary procedures like `notify` but there is no harm done. If this offends your sensibilities, you can be more specific about the exported subcommands.

The second task, a means of creating a `bureaucrat` channel, is already provided for us by Tcl with the `chan create` command. The *`MODE`* argument to the command is a list of one or two elements with values `read` or `write` that determine whether the channel is read-only, write-only or read-write. The *`CMDPREFIX`* parameter is the command prefix the Tcl I/O system should call to implement the various operations. Thus to create a `bureaucrat` channel, the application would call

```
set chan [chan create {read write} ::bureaucrat]
```

21.3.1.8. Seeking in a reflected channel

By its nature, just like the built-in sockets our reflected channel type is not amenable to seeking and thus will not actually implement the optional . However, we describe it here just so as to not get an incomplete grade.

The signature for the `seek` subcommand is

```
seek CHANNEL OFFSET BASE
```

The command is called to position the access pointer from where the next read or write will take place. The arguments are exactly those we described for the application level `chan seek` command (Section 13.15.2).

The subcommand return value should be the absolute position of the access pointer after it has been moved. The application level `chan tell` (Section 13.15.1) also maps to this `seek` subcommand. Tcl will call it with *`OFFSET`* set to `0` and *`BASE`* set to `current`, in effect getting the value of the access pointer without actually moving it.

21.3.1.9. The complete channel implementation

We can now put together our complete implementation, with some minor stylistic changes to put the procedures inside a namespace block.

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}

    proc initialize {chan mode} {
        variable channels
        set channels($chan) {
            State    OPEN
            Data     {}
            InFlight 0
            Delay    100
            Blocking 1
            Watch    {}
        }
        return {initialize finalize watch read write configure cget cgetall \
            blocking}
    }

    proc finalize {chan} {
        variable channels
        unset -nocomplain channels($chan)
    }

    proc configure {chan optname optval} {
        variable channels
        if {$optname ne "-delay"} {
            error "Unknown option \"$optname\"."
        }
        dict set channels($chan) Delay $optval
        return
    }

    proc cget {chan optname} {
        variable channels
        if {$optname ne "-delay"} {
            error "Unknown option \"$optname\"."
        }
        return [dict get $channels $chan Delay]
    }

    proc cgetall {chan} {
        variable channels
        return [list -delay [cget $chan -delay]]
    }
```

```
proc blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}

proc watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}

proc notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}

proc write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
        return 0
    }
    dict with channels($chan) {
        if {$InFlight} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
        }
        set InFlight 1
        after $Delay [list [namespace current]::delayed_receive $chan \
            $bytes] ❶
    }
    return [string length $bytes]
}
```

```
    proc delayed_receive {chan bytes} {
        variable channels
        if {! [info exists channels($chan)]} {
            return;
        }
        dict with channels($chan) {
            if {$State ne "OPEN"} {
                return
            }
            append Data $bytes
            set InFlight 0
        }
        notify $chan read
        notify $chan write
    }

    proc read {chan count} {
        variable channels
        dict with channels($chan) {
            if {[string length $Data] == 0} {
                if {$Blocking} {
                    return -code error EAGAIN ❷
                } else {
                    return -code error EAGAIN
                }
                return -code error EAGAIN
            }
            set bytes [string range $Data 0 $count-1]
            set Data [string range $Data $count end]
        }
        notify $chan read
        return $bytes
    }

    namespace export *
    namespace ensemble create
}
→ ::bureaucrat
```

❶ Postpone the actual write
❷ Same code as the false clause! See the text

21.3.2. Using reflected channels

Use of a reflected channel is along the same lines as a built-in Tcl channel. We demonstrate our `bureaucrat` channel here, more so to prove it works than to illustrate any new commands or techniques.

We start off creating the channel.

```
set chan [chan create {read write} ::bureaucrat] → rc22
```

Except for channels backed by terminal devices, Tcl defaults to full buffering whereas we would like line buffering. Also, we want to test event driven I/O and further verify our custom option `-delay` is effective. We configure the channel accordingly.

```
chan configure $chan -buffering line -blocking 0 -delay 1000
```

We set up a handler to be called when the channel is writable and implement it using an anonymous procedure that we define using our `lambda` helper (Section 3.5.9.4). Because of the artificial flow control we implemented, we expect it will be triggered every second or so based on the value we set for `-delay`. It then writes out the current time to the channel.

```
set counter 0
chan event $chan writable [lambda {chan} {
    if {[incr ::counter] > 2} {
        close $chan
        set ::done 1
        return
    }
    puts $chan [clock format [clock seconds] -format %T]
} $chan]
```

Similarly, our read handler prints out the time the message is received. Strictly speaking we do not need the end-of-file check since we are writing and reading into the same channel. When the writer closes the channel, no handlers will be called anyways once it is closed. But we include the check for completeness.

```
chan event $chan readable [lambda {chan} {
    set message [gets $chan]
    if {$message eq "" && [chan eof $chan]} {
        close $chan
        return
    }
    puts "Received \"$message\" at [clock format [clock seconds] -format %T]"
    return
} $chan]
```

Finally, assuming we are running in `tclsh` without the event loop active, we fire it up. As we hoped, messages are sent at one second intervals and received after our configured delay.

```
% vwait ::done
→ Received "21:05:38" at 21:05:39
  Received "21:05:39" at 21:05:40
```

21.3.3. Reflected channel limitations

There are some limitations in the functionality of reflected channels.

The first and most important is that reflected channels are not associated with operating system level descriptors or handles and are invisible to other processes or even other non-

Tcl threads. Therefore they cannot be passed to child processes, used for `exec` (Section 20.1) redirection and so on.

Secondly, reflected channels cannot support the half-closing of read-write channels.

The third limitation is more subtle and involve blocking operation. Moreover, it is only specific to reflected channel implementations in which the same channel was used for both reading and writing within a single Tcl thread. Since this is the type of reflected channel we used for illustrative purposes, we demonstrate with the following short example code.

```
set chan [chan create {read write} ::bureaucrat] → rc23
```

Note the channel is a blocking channel. Given it has no content, we expect the following `read` to permanently block. What happens instead is that we get an empty string back and if we call `chan block`, it indeed indicates no data is available.

```
read $chan 1       → (empty)
chan blocked $chan → 1
chan eof $chan     → 0
```

As per the Tcl reference page for `read`, for blocking channels our `read` should have only returned the empty string if the channel was at end-of-file. If no data was available, the command should have blocked.

This behaviour could be indeed be seen as a bug in our implementation. However, it is also exhibited by similar reflected channel implementations like `fifo` in Tcllib[5]. The issue, and reason why it cannot be fixed, is as follows. When our channel implementation's `read` subcommand is invoked and no data is available, there is no way for it to block **while still allowing the same thread to run and write to the channel to unblock it**. Use of commands like `vwait` for this purpose would violate blocking semantics. We are thus stuck between a rock and a hard place resulting in the seen behaviour.

It is worth reiterating that this flaw is only present in specific types of reflected channels. Those backed by a real operating system device, or one in which data is always available, such as the `random` channel in Tcllib that sources random byte sequences, will not have this blocking issue. Moreover, even for those that do, an application can work around is by checking `chan blocked` on empty results and not assuming end-of-file.

Nevertheless, these channels are best utilized in non-blocking event-driven mode.

[5] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

22

Networking and Communications

> Communication is everyone's panacea for everything.
>
> — Tom Peters

And what's true for humans is true for software. Communication with other systems is part of almost every modern application. This chapter covers Tcl support for communicating over TCP and serial ports.

22.1. Network communications

Tcl's built-in support for network communication only includes the transport layer TCP protocol which covers the vast majority of modern applications. Other transports like UDP and application protocols like HTTP are available through thirdparty packages[1] and not covered in this book.

The *Transmission Control Protocol* (TCP) is a stream oriented protocol where the data being transferred is seen as sequence of bytes without any message boundaries. This is a natural fit for Tcl's I/O model and allows the standard channel commands like `puts`, `read` etc. to be used for communicating over TCP as well.

22.1.1. Writing TCP clients: socket

```
socket ?-myaddr LOCALADDR? ?-myport LOCALPORT? ?-async? SERVER SERVERPORT
```

We will start off with the client end of TCP connections, describing the basic operation, asynchronous connects and various channel options specific to sockets.

The `socket` command is used by client applications to establish a TCP connection to a server. The command returns a read-write channel which can be used in the same fashion as was described for files and process pipes. *`SERVER`* is the DNS name or IP address (IPv4 or IPv6) of the server to which the connection is to be established. *`SERVERPORT`* is the port number on which the server is listening. In the case where a DNS name is passed that has multiple IP addresses associated with it, Tcl will try each in turn until one succeeds.

The address and port used for the local end of the connection will be automatically assigned by the system. If required, the application can bind the socket to a specific local address and port by passing the `-myaddr` and `-myport`.

[1] A HTTP package is part of the core distribution as well.

Like all channels, socket based channels must also be closed with the `chan close` command. Moreover, as was described for process pipelines (Section 20.4), it is also possible to "half-close" a channel by shutting down only one direction of the two-way connection.

22.1.1.1. Connecting synchronously

By default, the `socket` command will connect synchronously to the server and block until the connection is established.

Here is a simple example to retrieve a Web page.

```
% set http_req "GET http://www.example.com HTTP/1.1\n" ❶
→ GET http://www.example.com HTTP/1.1
% set so [socket www.example.com 80]
→ sock24156891260
% chan configure $so -encoding utf-8 -translation crlf -buffering line ❷
% puts $so $http_req
% close $so write ❸
% read $so
→ HTTP/1.0 400 Bad Request
  Server: AkamaiGHost
  Mime-Version: 1.0
...Additional lines omitted...
% close $so
```

❶ Note the extra newline to indicate end of HTTP headers
❷ HTTP is line-oriented with CR-LF line endings
❸ Half close the channel so server knows we are done

One point to be noted above is the half-closing of the channel. For HTTP 1.1, the server allows the client to send multiple requests. Our half-close indicates to the server that no more requests will be arriving, allowing it to then close its end of the connection. That in turn will result in an EOF on our end of the channel causing our `read` to complete. Without the half-close, our unconstrained `read` would block until the server timed out and closed its connection.

22.1.1.2. Connecting asynchronously

To avoid blocking until the connection is fully established, pass the `-async` option. The command will then initiate the connection but return right away before the connection is fully established.

As of the time of writing, if *`SERVER`* specifies a host name as opposed to an IP address, the `socket` command is not completely asynchronous as the name lookup is synchronous.

Note that even with `-async` specified, the channel is still in blocking mode so any attempt to read from the socket will block until the connection is established **and** data is received from the other end.

Most commonly, an asynchronous connect is followed by configuration of the channel to be non-blocking and registering read and write event handlers. When the connection is established, the write event is fired indicating that the channel may now be written. Conversely, the read handler will be invoked upon the arrival of data over the connection.

Let us rewrite our previous script in asynchronous style using non-blocking sockets. Like all our asynchronous and event driven I/O examples, this script also relies on the event loop and hence the `vwait` on the variable `done`. In a real application which already had the event loop active, this would not be required.

```
proc on_read so {
    set s [read $so]
    if {[string length $s] == 0} {
        if {[eof $so]} {
            close $so
            set ::done 1
        }
        return
    }
    puts $s
}

proc on_write so {
    puts $so "GET http://www.example.com HTTP/1.1\n"
    chan event $so writable {}
    chan close $so write
}

set so [socket -async www.example.com 80]
fconfigure $so -blocking 0 -encoding utf-8 -translation crlf -buffering line
chan event $so readable [list on_read $so]
chan event $so writable [list on_write $so]
vwait ::done
→ HTTP/1.0 400 Bad Request
  Server: AkamaiGHost
  Mime-Version: 1.0
...Additional lines omitted...
```

We have already seen similar code in Chapter 21 so we just point out the differences. The `on_write` handler will be called when the socket channel is ready to be written to, the first instance of which is when the connection to the remote server is established. It writes the `GET` request to the channel and then removes the write handler on the channel as we will not have anything more to write to it. We also close the socket for writes so the server knows we have nothing more to send.

When data arrives, the `on_read` handler is called which reads all data received so far and prints it, closing the socket on an end of file.

22.1.2. Writing TCP servers: socket -server

```
socket -server COMMANDPREFIX ?-myaddr LOCALADDR? SERVERPORT
```

We now turn our attention to the server end of a network connection. A server starts listening on a port using the form of the `socket` command shown above. The command returns a channel for a socket that is listening on port *`SERVERPORT`*. If *`SERVERPORT`* is `0`, the system will pick a random unused port whose value can be retrieved with the `-sockname` option to `chan configure`.

By default, incoming requests to the specified port on any of the local system's addresses will be accepted. The `-myaddr` option restricts accepted connections to those targeting a specific DNS name or IP address on the local system.

When a client's connection request is received, Tcl creates a **new** channel attached to the network connection for that client. It then invokes *`COMMANDPREFIX`* with three additional arguments:

- the new channel which is to be used to communicate with this client,
- the client's IP address, and
- the port number on the client end of the connection.

Note the distinctions between the original channel returned by the `socket` command and the new channel:

- All communication occurs over the client channels created in response to received connection requests. The channel returned by a `socket -server` command cannot be used for any data transfer.
- Closing a client channel does not impact the listening channel. New connections will continue to be accepted. Likewise, closing the listening channel will prevent new connections from being accepted but will not impact existing connections.
- The channel options, such as `-buffering` etc., have to be set individually for each client channel. They are **not** inherited from the listening socket.

Here is an example server that services multiple clients. It assumes a line oriented protocol and responds to each line from a client with the client's port number followed by the received line with the characters reversed.

We start off by defining the command that will invoked in response to every connection request. We want to be able to serve multiple clients so we make the accepted client channel non-blocking and attach a read handler.

```
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation \
        crlf
    chan event $so readable [list on_read $so $client_port]
}
```

Our read handler reverses the line and sends it back to the client. As a special case, an empty line from the client will end our server.

```
proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0 ❶
    } elseif {[eof $so]} {
        close $so
    }
}
```

❶ Treat empty line as exit server command (**any** client can shut down server)

Now we create the listening socket on a port. Instead of picking a port ourselves, we will let the system pick one for us by specifying `0` as the port number. Also we will only listen on our local loopback address. After creating the listening socket the server will sit waiting in the event loop.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 49999]
vwait forever
```

Here is the entire server script.

```
# server.tcl
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation \
        crlf
    chan event $so readable [list on_read $so $client_port]
}

proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0
    } elseif {[eof $so]} {
        close $so
    }
}

set listener [socket -server on_accept -myaddr 127.0.0.1 49999]
vwait forever
```

Let us run the server **in a separate process** from an interactive client.

```
exec [info nameofexecutable] scripts/server.tcl & → 15808
```

When connecting, we take care to configure the client channels with the same options as the server end.

```
set so [socket 127.0.0.1 49999]                                          →
 sock241549e9da0
chan configure $so -buffering line -encoding utf-8 -translation crlf → (empty)
puts $so abc                                                             → (empty)
gets $so                                                                 → 22929: cba
puts $so ""                                                              → (empty) ❶
close $so                                                                → (empty)
```

❶ Shut down the server

22.1.3. Socket configuration

The `chan configure` command supports additional settings for socket based channels. These are shown in Table 22.1.

Table 22.1. Socket-specific configuration options

Option	Description
`-connecting`	Returns `1` if the socket is still in the process of connecting and `0` otherwise. This option is only supported for client side sockets and is intended to be used for asynchronous connects that specify the `-async` option.
`-error`	Returns any error status currently associated with a socket.
`-keepalive`	Sets or retrieves the `SO_KEEPALIVE` system socket option that enables sending of keep-alive messages. Not available in Tcl 8.6 and earlier.
`-nodelay`	Sets or retrieves the `TCP_NODELAY` system TCP socket option. A true value **disables** the Nagle algorithm; a false value enables it. Not available in Tcl 8.6 and earlier.
`-peername`	Returns a list of three elements containing the address, hostname, and port of the remote end of a connection. This option is not supported for listening sockets as they are not connected.
`-sockname`	For connected sockets this returns a list of three elements containing the address, hostname and port of the local end of a connection. For listening sockets, the return value is a flat (i.e. not nested) list that may contain more than one such triple corresponding to each address and port that is being listened on.

Some example uses of the options are below.

Checking for the state of an asynchronous connection:

```
set so [socket -async 127.0.0.3 49998] → sock24154b4bef0
chan configure $so -connecting         → 1
```

Retrieving the remote end point of a connection:

```
% set so [socket www.microsoft.com 80]
→ sock000000000310BAC0
% chan configure $so -peername
→ 23.15.107.229 a23-15-107-229.deploy.static.akamaitechnologies.com 80
% close $so
```

Retrieving the addresses and ports for a listening socket:

```
set so [socket -server accept 49998] → sock24154a97fd0
chan configure $so -sockname         → 0.0.0.0 0.0.0.0 49998 :: :: 49998
```

22.1.3.1. Text and binary protocols

It is particularly important to ensure the channel configuration settings are correct for the application protocol used over a network connection. Obviously, both the client and server ends must use the same settings.

Binary protocols

For binary application protocols that treat data as a sequence of bytes, the `-translation` channel option must be set to `binary`. See Section 13.14 for details.

```
chan configure $chan -translation binary
```

Needless to say, the strings written to the channel must also be binary strings (Chapter 8).

Text protocols

For text based protocols, the channel options to consider are `-translation`, `-encoding` and, rarely, `-eofchar`.

- The application protocol defines the character sequence for line endings and accordingly the `-translation` option must be set to `lf`, `cr` or as is most common, `crlf`.
- The default `-encoding` on the created channel is the system encoding. This is rarely correct since the systems at the two ends could very well be using different system encodings. Most modern protocols use UTF-8.
- For text protocols that are **line** oriented, set the `-buffering` option to `line` mode or remember to do a `chan flush` on the channel at appropriate times.
- Application protocols do not generally use character sequences to indicate end of file so the `socket` command's default of disabling this feature is usually appropriate.

22.1.4. The http package

The `http` package provides client-side support for the ubiquitous HTTP protocol for retrieving Web pages. While bundled with Tcl, the functionality is not really part of the core Tcl language and is only illustrated here with a very brief example.

Web pages are retrieved by passing their URL to the `http::geturl` command.

```
package require http                          → 2.10.0
set tok [http::geturl http://www.example.com] → ::http::1
```

The command returns a token that is passed to other `http` package commands to retrieve the status of the request and the server response.

```
% http::status $tok ❶
→ ok
% http::responseLine $tok ❷
→ HTTP/1.1 200 OK
% http::responseHeaders $tok ❸
→ accept-ranges bytes content-type text/html etag {"84238dfc8092e5d9c0dac8ef933...
% http::responseBody $tok ❹
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

...Additional lines omitted...
% http::cleanup $tok ❺
```

❶ Completion status of the request.
❷ The status response line from the server.
❸ A dictionary holding the response headers.
❹ The actual web page content.
❺ Free up resources.

Secure connections over the HTTPS protocol require the use of additional Tcl extensions implementing the TLS protocol, e.g. TWAPI[2] (Windows only) or TclTLS[3]. The HTTPS protocol and the command to create a TLS channel have to be registered with the `http::register` command. Retrieval of the page then follows the same sequence as above.

```
% package require twapi
→ 5.0.2
% http::register https 443 twapi::tls_socket
→ 443 twapi::tls_socket {} 0 0
% set tok [http::geturl https://www.example.com]
→ ::http::2
% http::status $tok
→ ok
% http::responseBody $tok
→ <!doctype html>
  <html>
...Additional lines omitted...
% http::cleanup $tok
```

[2] https://twapi.magicsplat.com
[3] https://github.com/bohagan1/TclTLS

The package includes many other commands and options dealing with the different modes of operation of the HTTP protocol. See the reference documentation for details.

22.2. Communication over serial ports

```
open PATH ?ACCESS? ?PERMISSIONS?
```

Before the Internet and Gigabit desktop connections, there were serial ports. Even today, serial ports are still used in a wide variety of devices ranging from the humble RS-232 console ports still available on some servers to industrial control equipment. Tcl's channel-based I/O system includes support for communicating over these interfaces.

A channel to a serial port is opened with the same `open` command we described for opening disk based files. Here *`PATH`* identifies a serial port. The *`ACCESS`* and *`PERMISSIONS`* parameters are exactly as we described for files in Section 13.3 so we will not describe them here. Similarly use of `puts` for output, `gets` and `read` for input and event driven I/O follows the same patterns we have described for the various channel types. We will therefore stick only to aspects like configuration options that are specific to serial ports. Familiarity with serial port terminology and operation is assumed.

On Unix/Linux systems, serial port channels are opened by specifying a file system path that maps to a serial port. This is usually, but not necessarily of the form `/dev/ttyN` where *`N`* is the serial port number. On Windows systems, serial ports are specified either in the form `COMN:` where *`N`* is a serial port number in the range `1-9` or in the form `//./comN` where *`N`* is **any** serial port number. Platforms other than Windows and Unix may use some other syntax to identify serial ports.

22.2.1. Serial port buffer and queue sizes

On Windows and Unix, the number of bytes in the input and output queues can be obtained with the read-only `-queue` option. This returns a pair containing the number of characters present in the input and output queues.

On Windows only, the `-sysbuffer` option sets the maximum size of the system buffers. The value is a list of one or two integers. The first specifies the size of the input buffer and the second, if present, the size of the output buffer.

22.2.2. Serial port speed, parity, and bit lengths

The `-mode` option to `chan configure` is used to modify or retrieve the baud rate, parity bits, data bits and stop bits settings for a serial port. The argument to the option consists of 4 values separated by commas in the form *`SPEED`*`,`*`PARITY`*`,`*`DATABITS`*`,`*`STOPBITS`* corresponding to those four settings. The *`PARITY`* value should be one of `n`, `o`, `e`, `m` or `s` signifying none, odd, even, mark or space respectively. The *`DATABITS`* value should be a number in the range `5-8`. The *`STOPBITS`* value should be either `1` or `2`.

22.2.3. Serial port flow control

Serial ports may use one of several handshaking mechanisms for implementing flow control to ensure input buffers are not overrun. The `-handshake` option to `chan configure` controls the mechansim to use. The configured value should be one from Table 22.2.

Table 22.2. Values for handshake configuration

Value	Description
`none`	Turns off any form of flow control.
`rtscts`	Enables flow control using RTS and CTS control signals.
`dtrdsr`	*Windows only*. Enables flow control using the DTR and DSR control signals.
`xonxoff`	Specifies software based handshake. The pair of characters used for flow control can be then be specified with the `-xchar` option. The value should be a pair of characters used to enable and disable data transfer. The default value is the ASCII standard XON/XOFF pair.

The handshake can also be implemented by directly setting the state of the RTS and DTR control lines and querying the status of the CTS, DSR, RING and DCD lines.

The `-ttycontrol` option controls the state of the outgoing control lines, RTS and DTR. It is recommended that you not directly control these when the `-handshake` option is set to `rtscts` or `dtrdsr`. The value for the option is a dictionary keyed by the signal names `RTS` and `DTR` with the corresponding element value set to `0` or `1` to turn the signal off or on.

The `-ttycontrol` option also permits setting or resetting the BREAK condition on the serial port by setting the value of the `BREAK` key in the dictionary passed as the option value.

Conversely, detection of the input control signals is done with the `-ttystatus` option. This returns a dictionary with the keys `CTS`, `DSR`, `RING` and `DCD` set to `0` or `1` reflecting the signal states.

22.2.4. Timers related to serial ports

There are two timers that can be configured for serial port channels. The first of these is available for both Windows and Unix and set through the `-timeout` option. This sets the interval after which a blocking read operation will time out. The option value is specified in milliseconds. The granularity of the timer value is platform dependent.

The other timer related option is `-pollinterval` and is only applicable to Windows systems. It controls the maximum time between polling for file events. **Setting this value lower than the default 10ms Tcl uses to check for all types of events will also increase the frequency of the latter.**

22.2.5. Checking for serial port errors

The read-only `-lasterror` option returns more detailed error information for serial ports than the standard error codes used by Tcl's file I/O commands. Examples include receive buffer overruns, framing errors, etc. See the reference documentation for possible values, their causes and potential remedies.

23

Interpreters

A *Tcl interpreter* runs Tcl programs within an execution context that includes a namespace hierarchy, command and variable definitions, and a call stack. Applications create a new Tcl interpreter through a native call to the Tcl library as part of their initialization. Some applications may even create multiple such interpreters from native code, either in the same thread or from different threads. These interpreters all run independently, oblivious to the others for the most part.

All our discussion so far has pertained to running Tcl within a single such interpreter created by the application. In fact, an interpreter may **itself** create additional *child* interpreters. Unlike the interpreters created natively by the application, these child interpreters are **not** independent. They run under the control of the creating interpreter, their *parent*, in terms of the commands they may execute, how long they may run for and so on. Each child may in turn create additional interpreters, resulting in an hierarchy of interpreters.

Multiple interpreters are useful in many situations. We will see two examples in this chapter — a network server that uses a separate interpreter for every client, thereby avoiding even accidental interference between the contexts for each client, and the use of multiple interpreters in implementation of domain-specific languages.

A *safe* interpreter is a special kind of interpreter where commands deemed to be dangerous from a security point of view are not available. This is useful to allow execution of code from unknown and untrusted sources. An example is the execution of application plug-ins written by the user or third parties in a safe interpreter so as to avoid both inadvertent as well as malicious modification of the application code.

23.1. Creating interpreters: interp create

```
interp create ?-safe? ?--? ?CHILDPATH?
```

An interpreter is created with the `interp create` command. Specifying the optional `-safe` option results in the creation of a safe interpreter. Safe intepreters are discussed in Section 23.10. The `--` sequence indicates the end of options.

If *`CHILDPATH`* is not specified, the command creates an interpreter as a direct child of the current interpreter (i.e. the interpreter invoking the `interp create` command). The name of the child is automatically generated.

```
set child1 [interp create] → interp0
```

The return value from the command is the name of the newly created interpreter. In addition to creating the interpreter, a new command of the same name is also created in the **current** interpreter in the global namespace. This proxy command can be used to in various ways to evaluate code or otherwise control the child.

23.2. Identifying interpreters

An interpreter is identified through a list of interpreter names that define a path through the interpreter hierarchy relative to the current interpreter. The *`CHILDPATH`* argument may be specified to the `interp create` command to create a new interpreter at any level in the hierarchy **below** the current interpreter. The last element of this path is the name of the new interpreter. For example, we can create another direct child of the current interpreter.

```
set child2 [interp create secondchild] → secondchild
```

Since the list passed as the *`CHILDPATH`* argument contains only one element, it becomes the name of the created child interpreter. We can also create interpreters deeper in the hierarchy.

```
set grandchild1 [interp create [list $child1 grandchild]] → interp0 grandchild
set grandchild2 [interp create {secondchild grandchild}]  → secondchild grandchild
```

Both grandchildren have the same name but are distinguished because their paths are different. The new interpreters are created in their **parent**, **not** in the interpreter that invoked the `interp create` command.

Note that all interpreters except the last in the path must already exist.

Interpreter paths are **always** relative to the current interpreter. There is no concept of an “absolute” path and no way to reference an interpreter that is not a descendant of the current interpreter. An empty path is generally treated as referring to the current interpreter.

As is the case when *`CHILDPATH`* is unspecified, a new command of the same name as the created interpreter is also created in its parent. If a command of that name already exists in the parent, it is overwritten. If the name contains one or more namespace separator sequences, the command is defined within that namespace which is created if necessary.

```
interp create ns::ip  → ns::ip ❶
interp children       → secondchild ns::ip interp0
ns::ip eval {set x 1} → 1
```

❶ Creates a child `ns::ip` and command `ip` within namespace `ns`.

In such cases, deleting the namespace will also delete the interpreters within it.

```
namespace delete ns → (empty)
interp children     → secondchild interp0
```

23.3. Inspecting the interpreter hierarchy: interp children | exists

```
interp children ?INTERPPATH?
interp exists INTERPPATH
```

The `interp children` command returns the names of the interpreters that are direct children of *`INTERPPATH`*. If *`INTERPPATH`* is not specified, it defaults to the current interpreter.

```
interp children                          → secondchild interp0
interp children {}                       → secondchild interp0 ❶
interp children secondchild              → grandchild
interp children {secondchild grandchild} → (empty) ❷
```

❶ Empty path refers to current interpreter
❷ No fourth generation

The `interp exists` command checks for the existence of a specific descendent with a known path, returning `1` if a specified interpreter exists and `0` otherwise.

```
interp exists {secondchild grandchild}                → 1
interp exists {secondchild grandchild greatgrandchild} → 0
```

23.4. Destroying interpreters

The `interp delete` command destroys one or more interpreters.

```
interp delete ?INTERPPATH …?
```

Each *`INTERPPATH`* argument identifies a descendent of the current interpreter. Deleting an interpreter will also destroy that interpreter's descendents. If any specified interpreter does not exist, an error is raised and any interpreters listed later in the argument list will not be destroyed. Deleting an interpreter also deletes the command of that name.

```
interp children           → secondchild interp0
interp delete secondchild → (empty)
interp children           → interp0
```

23.5. Evaluating scripts in an interpreter: interp eval

```
interp eval INTERPPATH ARG ?ARG …?
INTERPCMD eval ARG ?ARG …?
```

We saw in Section 3.13 the use of the `eval` command to execute scripts. The `interp eval` command is similar except that it allows execution of the script in **any** accessible interpreter by specifying its path.

The command concatenates all *ARG* arguments in the same fashion as the `eval` command and then executes the script in the interpreter identified by *INTERPPATH*. The command below will set the value of the `foo` variable in the interpreter identified by `$child1`.

```
interp eval $child1 { set foo 1 } → 1
```

An alternative means of achieving the same result is to use the `eval` subcommand of the target interpreter's proxy command. The following is equivalent to the above.

```
$child1 eval {set foo 1} → 1
```

In both cases, the result is the result of the script evaluation.

Like the `eval` command, the arguments undergo double substitution (Section 3.13.1). However, in this case, the two rounds of substitution take place in separate interpreters. The first round of substitution happens place in the current interpreter and the second in the specified target interpreter. The following snippet illustrates this.

```
set bar foo             → foo
set foo 2               → 2
$child1 eval set foo 1 → 1
eval set $bar           → 2
$child1 eval set $bar   → 1
```

23.6. Command aliases

A command *alias* maps a command in an interpreter to a command defined in another interpreter. Invoking the command in the first will execute the mapped command in the context of the second.

23.6.1. Defining aliases: interp alias

```
interp alias SRCINTERP SRCCMD TARGETINTERP TARGETCMD ?ARG …?
INTERPCMD alias SRCCMD TARGETCMD ?ARG …?
```

The `interp alias` command defines a command alias. The command creates a new alias named *SRCCMD* in the interpreter whose path is given by *SRCINTERP*. When *SRCCMD* is invoked within *SRCINTERP*, Tcl will execute *TARGETCMD* in interpreter *TARGETINTERP*. The arguments passed to *TARGETCMD* will consist of the *ARG* arguments, if any, specified in the `interp alias` command, followed by any arguments specified in the invocation of *SRCCMD*.

Another way to create aliases is with the `alias` subcommand of the interpreter's proxy command as shown in the second syntactic form above. This works similarly except that the target interpreter in which *TARGETCMD* will run, is the current interpreter.

The result of the command in both forms is an alias token which can later be used to introspect or delete the alias. Although the returned result may appear to be what was passed as *SRCCMD*, **this need not be the case**. The token remains unchanged even if the alias is renamed.

The primary motivation for using aliases is to execute commands in controlled fashion on behalf of a “less privileged” interpreter. Section 23.10 describes this in detail; here we stick to a couple of basic examples.

One of the simpler uses of aliases is as a short hand for lazy programmers who hate typing.

```
% interp alias {} xmlify {} string map {< &lt; > &gt; & &amp; \" &quot; ' &apos; }
→ xmlify
% xmlify "2 < 3"
→ 2 &lt; 3
```

Note from the above example that

- An empty path passed as the source or target interpreters refers to the current interpreter.
- You can create an alias within the same interpreter, i.e. where the target and source interpreters are the same.

Of course, you could use procedures for the above as well so here is an example where you do need aliases. Imagine our server application dedicates an interpreter to each client and also has a common logging system shared by all interpreters. We define an interpreter, `logger`, to collect these messages.

```
interp create logger
interp eval logger {
    set ::log_level 1
    proc log {level message} {
        if {$level >= $::log_level} {
            lappend ::messages $message
        }
    }
    proc messages {} {
        return $::messages
    }
}
```

Then we define aliases in each client interpreter to send messages to it.

```
interp create client0
interp alias client0 debug logger log 0
interp alias client0 log logger log 1
interp alias client0 err logger log 2
client0 eval {
    debug "This is a debug message."
    log "This is an informational message."
    err "This is an error message."
}
logger eval messages
→ {This is an informational message.} {This is an error message.}
```

23.6.2. Deleting aliases

```
interp alias INTERPPATH ALIASTOKEN {}
INTERPCMD alias ALIASTOKEN {}
```

An alias defined in an interpreter may be deleted using one of the above command forms. Note `ALIASTOKEN` is the token returned when the alias was created.

```
set ip [interp create]                  → interp1
set token [$ip alias aliased_puts puts] → aliased_puts
$ip eval aliased_puts Hullo!            → Hullo!
interp alias $ip $token {}              → (empty) ❶
$ip eval aliased_puts Hullo!            Ø invalid command name "aliased_puts"
```

❶ Delete the alias.

The primary purpose of this simple snippet is to stress that aliases are managed with the token returned at the time they were created. This token is not guaranteed to be the same as the alias itself. Thus to delete the token, we passed `$token`, not `aliased_puts`, and did not assume they would be the same.

23.6.3. Introspecting aliases: interp aliases|target

```
interp aliases INTERPPATH
INTERPCMD aliases
interp alias INTERPPATH ALIASTOKEN
interp target INTERPPATH ALIASTOKEN
```

The `interp aliases` command or the `aliases` subcommand of its proxy command return the list of alias tokens defined in an interpreter.

```
interp aliases client0 → err log debug
client0 aliases        → err log debug
```

The target interpreter for an alias can then be determined with the `interp target` command or its proxy form.

```
interp target client0 debug → logger
```

The command will return an empty string if the current interpreter is the target and raise an error if the alias is not defined.

The command prefix to which an alias resolves in the target interpreter can be retrieved with another syntactic form of `interp alias` or its proxy equivalent.

```
interp alias client0 debug → log 0
client0 alias err          → log 2
```

23.7. Execution context in child interpreters

When a script is executed via `interp eval`, it is evaluated in the **current** context of the target interpreter, **not** its global context. The following snippet illustrates this.

```
set ip [interp create]
proc get_child_context {ip} {
    return [$ip eval {namespace current}]
}
$ip alias context_demo get_child_context $ip
$ip eval { namespace eval ns {context_demo} }
→ ::ns
```

Note how the `namespace current` invocation from `get_child_context` shows `ns` as the namespace context and not the `::` global context. The `context_demo` alias is invoked in the `ns` namespace context within the child. This results in the `get_child_context` procedure being run in the **parent**. When that evaluates `namespace current` in the **child**, that command is then invoked within the child's current context, which is `ns`.

As another example, consider the invocation of an alias from within a procedure.

```
proc get_child_var {ip} {
    return [$ip eval {set myvar}]
}
$ip alias var_demo get_child_var $ip
$ip eval {
    set myvar "global"
    proc demo {} {
        set myvar "local"
        var_demo
    }
}
$ip eval demo
→ local
```

Again, note how the value returned is `local` indicating that the `set myvar` was executed in the **current** context of the child, i.e. within the `demo` procedure context, not the global one.

23.8. Cancelling script evaluation: interp cancel

```
interp cancel ?-unwind? ?--? INTERP ?MESSAGE?
```

The `interp cancel` command aborts a script running in an interpreter. The effect is similar to that of throwing an error from within a script executing in the interpreter identified by the interpreter path *INTERP*. The difference is that the cancellation can be initiated from outside the script itself, and unlike error exceptions, the cancellation can be made to be untrappable with `catch` and `try`.

Let us examine the effect of this command by experimenting with the current interpreter itself. First we define three commands that will raise an exception, or call `interp cancel` on the current interpreter.

```
proc raise_error {} { error "Error raised!" }
proc cancel {} { interp cancel {} "Interpreter execution cancelled!" }
proc cancel_unwind {} {
    interp cancel -unwind -- {} "Interpreter execution unwound!"
}
```

Then we define a `demo` procedure that will call one of these procedures based on the argument passed. We will add a variable trace to further distinguish the cases.

```
proc demo {procedure} {
    puts "Entering demo"
    set x 1
    trace add variable x unset print_args
    catch $procedure message
    puts "Caught error message: $message"
    puts "Exiting demo"
}
```

Now let us try calling `demo` with each of the procedures we defined earlier.

```
% demo raise_error
→ Entering demo
  Caught error message: Error raised!
  Exiting demo
  Args: x, , unset
```

We do not have much to say for the above case as we have already discussed raising of errors extensively in Section 15.5.

The `interp cancel` command without the `-unwind` option behaves very similarly.

```
% demo cancel
→ Entering demo
  Caught error message: Interpreter execution cancelled!
  Exiting demo
  Args: x, , unset
```

Again, the error is caught and the variable trace fires as the procedure is exited. The real difference with the use of `interp cancel` compared to `error` is that the exception can be initiated from a **different** interpreter than the one in which the script is running.

Our final example uses the `interp cancel` command with the `-unwind` option.

```
% demo cancel_unwind
Ø Entering demo
  Interpreter execution unwound!
```

When the `-unwind` option is specified, the call stack is completely unwound in the target interpreter **without any chance of trap or trace handlers to run** as seen in the example. Because the `-unwind` option does not give any error trap or trace handlers a chance to run, the common use of these handlers to release resources will be bypassed. Thus the mechanism should be used with extreme care and under limited circumstances.

Note that `interp cancel` aborts the execution of scripts in the target interpreter, the target interpreter itself is **not** deleted.

Our interpreter demonstrated the workings of `interp cancel` using the current interpreter. More commonly it is used in two scenarios:

- cancel scripts in a child interpreter from within an aliased command. Applying the above discussion to this case should be straightforward.
- cancel scripts running in a interpreter within another thread.

23.9. Sharing channels in interpreters

```
interp share FROMINTERP CHAN TOINTERP
interp transfer FROMINTERP CHAN TOINTERP
```

Channels created in an interpreter are not automatically available in other interpreters except for `stdin`, `stdout` and `stderr` which are present in all.

Channels can be made available in interpreters other than the one that created them:

- The `interp transfer` command makes a channel available in an interpreter while removing it from the original owner.
- The `interp share` command makes a channel available in an interpreter without removing it from the original owner.

Here *`FROMINTERP`* is the interpreter currently holding the channel *`CHAN`*, and *`TOINTERP`* is the interpreter where the channel is to be made available. The channel name remains the same in all interpreters where the channel is accessible.

In the case of shared channels, the seek pointer is shared across all interpreters so if multiple interpreters perform I/O on the shared channel the data will be interleaved. A channel is closed only when closed by all interpreters where it is available.

```
set ip [interp create]                              → interp3
set fd [file tempfile]                              → file2415639f510
interp share {} $fd $ip                             → (empty)
$ip eval "set fd $fd"                               → file2415639f510 ❶
$ip eval {puts $fd "Hullo from child"; close $fd}   → (empty) ❷
chan seek $fd 0 start                               → (empty) ❸
read $fd                                            → Hullo from child
close $fd                                           → (empty)
```

❶ Need to pass in the channel name to the child.
❷ File closed in child ...
❸ ... but still open in parent.

On Windows, GUI applications like `wish` do not have real standard I/O file descriptors. Standard I/O is simulated to a pseudo-console window provided by the Tk GUI extension, not implemented as real channels, and are therefore only available within the main Tcl interpreter.

23.10. Safe interpreters

There are many applications that need to run source code from untrusted sources.

- Browsers running Javascript downloaded from some random web site.
- Applications allowing thirdparty plug-ins.
- Privileged mode applications executing requests from an unprivileged user.
- Distributed computing tasks that need to be isolated from local system resources.

Tcl's solution for dealing with these situations is the *safe interpreter* which is a Tcl interpreter from which certain commands, which are considered dangerous when invoked from untrusted scripts, have been hidden.

A safe interpreter has the following restrictions placed on it:

- Commands like `exec`, `open`, `socket` etc. that can access system resources or affect the state of a process are not available.
- Safe interpreters cannot create unsafe interpreters.
- The global variable `env` is not available.
- Tcl's default auto-loading facilities (Section 3.5.1.1) are not implemented.
- Binary extensions cannot be loaded unless they have a specific entry point intended for safe interpreters.
- A safe interpreter cannot change its recursion limit (Section 23.11).

A completely restricted safe interpreter can be too limited in functionality to be useful in many circumstances. It can essentially do operations on data and little else. Tcl therefore provides mechanisms for permitting controlled execution of hidden commands in safe interpreters:

- Command aliases in the context of safe interpreters (Section 23.10.2);
- Invocation of hidden commands under the control of its parent (Section 23.10.3).

23.10.1. Creating a safe interpreter

A safe interpreter is created by passing the `-safe` option to `interp create`.

```
interp create -safe safe0 → safe0
```

We can verify that certain commands are **hidden** and therefore not directly available in the safe interpreter. Attempting to invoke them will result in an error.

```
% safe0 eval {socket www.example.com 80}
Ø invalid command name "socket"
```

See the reference documentation for `interp` for a complete list of the commands that are hidden in safe interpreters by default. Commands can be hidden dynamically at runtime and you can programmatically retrieve the list of currently hidden commands (Section 23.10.3).

Hidden commands are not actually removed from the interpreter and can be made available to the interpreter under the control of its parent.

23.10.2. Aliasing in safe interpreters

Removing **all** access to resources such as the file system would make safe interpreters usable in very few scenarios. For example, a safe interpreter in a web server handling client connections needs to be able to read Web content, write to log files and so forth while being prevented from accessing files from outside the Web content directories.

One mechanism available to achieve this goal of providing controlled access to resources is command aliases (Section 23.6). Our short illustration below demonstrates how a safe interpreter might write log messages to a file. Since the safe interpreter cannot directly open files, the trusted parent must do it on its behalf by aliasing a `log_message` command in the **child** to a command in the **parent** that does the actual write to the log file.

```
set safe_ip [interp create -safe]
proc log_message {message} {
    set log_chan [open [file join /var/log server.log] a]
    puts $log_chan $message
    close $log_chan
}
$safe_ip alias log_message log_message
```

Now scripts running in the child interpreter can safely log messages to disk while still being restricted in terms of its ability to do I/O.

```
$safe_ip eval { log_message "Something happened." }
```

The above example enabled some very specific functionality within the safe interpreter. More often you want the safe interpreter to have a little more generalized capabilities. For example, the safe interpreter serving Web clients has to be allowed to read any file in the Web content directory based on the URL but no other. It may also permit writing of files uploaded by the user but again only to a specific directory. We might thus choose to expose a restricted form of the `open` command to the safe interpreter. In **pseudocode** form,

```
$safe_ip alias open SafeOpen $child
proc SafeOpen {child path access} {
    if {![path_is_allowed $path $access]} {
        error "Access denied!"
    }
    tailcall $child invokehidden -- open $path $access
}
```

The code maps the `open` command in the safe child interpreter to the `SafeOpen` command in the parent passing the child interpreter command as an additional parameter. When the

```
open index.html r
```

command is invoked in the **child**, the **parent** runs the command

```
SafeOpen interp0 open index.html r
```

The `SafeOpen` command first checks whether the desired access to the path is permitted (we will elaborate on this aspect in a later section). If not permitted, it raises an error. Otherwise, it opens the specified file in the child interpreter with the `invokehidden` command that we describe a little later. The child can then read and / or write that file as appropriate.

Commands that create channels (`open` , `socket` etc.) are hidden by default in safe interpreters. However, commands that perform I/O on channels (`gets` , `puts` etc.) are available so they can be used on channels shared by the parent (Section 23.9).

23.10.2.1. Precautions for aliased commands

Commands like `SafeOpen` that are aliased into safe interpreters must be very careful in how they handle any arguments passed by the child interpreters. To safeguard against a malicious script running in the untrusted safe interpreter tricking the parent into executing arbitrary code, these arguments must **never** be treated as scripts or parts thereof, for example by passing them to commands like `eval` , `subst` , `expr` etc.

Also keep in mind that any exceptions raised in the parent interpreter will be propagated back to the child where they can be caught and error stack examined. Therefore if you want to prevent inadvertent information leakage from the parent to the child, all errors must be caught and passed on to safe interpreter in some sanitized generic form.

23.10.3. Hidden commands

Having looked at the use of aliases with safe interpreters, let us now look at the other mechanism for using safe child interpreters under the control of a trusted parent — hidden commands.

23.10.3.1. Invoking hidden commands: interp invokehidden

```
interp invokehidden CHILD ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG …?
CHILD invokehidden ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG …?
```

Commands that are hidden in an interpreter are **not** deleted or removed; they just cannot be directly invoked from within the safe interpreter. They can however be invoked by the **parent** through the `interp invokehidden` command.

Here the *`HIDDEN`* argument is the name of the command that is hidden in the child interpreter and *`ARG…`* are any arguments to be passed to it. The command is executed in the current

context of the child unless the `-global` and `-namespace` option is passed to force the execution to take place in the child's global or specified namespace instead.

We saw an example using `invokehidden` in the previous section where the `open` command was invoked in the child interpreter to create a channel to a file after the parent checked that the file path was one that the child was permitted to access.

```
tailcall $child invokehidden -- open $path $access
```

This results in the `open` command being run in the **child** interpreter and returning a channel to the script in the child.

We could have also written the above `invokehidden` command as

```
$child invokehidden -- open $path $access
```

The purpose behind `tailcall` is that in case of errors, the error stack looks cleaner as the `SafeOpen` call will not be present on the stack at all as the `tailcall` replaces it with the `open` command invocation.

At this point the front benchers in the class will be objecting vociferously, waking up the back bench in the process. Since the `open` command in the safe child interpreter is aliased to `SafeOpen` in the parent, won't the `invokehidden` call to `open` in the child result in a recursive call back into `SafeOpen` ad infinitum? The answer is that a call to `invokehidden` will always invoke the **original** command that was hidden and not any alias or redefinition. Here is an illustrative example.

```
% safe0 eval { proc demo {} { return "The original demo" } }
% safe0 eval demo
→ The original demo
% safe0 hide demo
% safe0 eval demo ❶
Ø invalid command name "demo"
% safe0 eval { proc demo {} { return "The redefined demo" } } ❷
% safe0 eval demo ❸
→ The redefined demo
% safe0 invokehidden demo ❹
→ The original demo
```

❶ Errors because the command is now hidden
❷ Redefine the procedure
❸ The redefinition is executed.
❹ The original hidden procedure is executed

Notice how invoking `demo` within the `safe0` interpreter runs the new definition of `demo` whereas invoking `demo` from the parent via `invokehidden` runs the original hidden procedure. This behaviour ensures that the child interpreter (remember it is potentially running untrusted code) cannot trick the parent into running the wrong code by simply redefining commands.

Like `interp eval`, `interp invokehidden` also runs code in the child interpreter. One obvious difference is that `interp eval` runs arbitrary scripts whereas `interp invokehidden` runs a single command which must furthermore be a hidden command. A less obvious but important difference is that while the arguments supplied to `interp eval` undergo two rounds of substitutions, once in the parent and once in the child, arguments to `interp invokehidden` only undergo substitution in the parent interpreter. This gives the parent precise control over the arguments that are passed to the hidden command.

23.10.3.2. Hiding and exposing commands: interp hide|expose

```
interp expose CHILD HIDDENNAME ?CMDNAME?
interp hide CHILD CMDNAME ?HIDDENNAME?
CHILD expose HIDDENNAME ?CMDNAME?
CHILD hide CMDNAME ?HIDDENNAME?
```

We noted earlier that certain commands are hidden at the time a safe interpreter is created. However, as our previous example showed, we can hide any command at any time with the `interp hide` command.

Both forms hide *`CMDNAME`* in the specified child interpreter. The command can be invoked by the parent via `invokehidden` with the name *`HIDDENNAME`* which defaults to *`CMDNAME`* if unspecified.

There is an important restriction that needs to be kept in mind with respect to the hiding of commands. Neither *`CMDNAME`*, nor *`HIDDENNAME`* if specified, can contain namespace qualifiers. The commands to be hidden are always looked up in the global namespace of the child interpreter. This restriction prevents the child from tricking the parent interpreter into hiding the wrong command by executing in a namespace other than the global one.

The `interp expose` command makes hidden commands available to the child interpreter. Exposing a command makes it available to be called directly by the child either under its original name or if *`CMDNAME`* is specified, under a new name. This technique is an alternative to renaming the original command when redefining it.

```
safe0 expose demo original_demo → (empty)
safe0 eval demo                 → The redefined demo
safe0 eval original_demo        → The original demo
```

One final note about hiding and exposing commands is that, although our discussion has been focused on safe interpreters, commands can be hidden or exposed in **any** child interpreter, not just safe ones, though their primary use is with the latter.

Here is an example of how the combination of aliases and hidden commands may be put to good use in contexts other than safe interpreters. In previous chapters, we saw several ways of “wrapping” existing commands to modify their behaviour. For example, we might want to log the every outgoing network connection in an application. We could create a new `socket` procedure that overrides the `socket` command and calls it. Here is a simple alternative.

```
interp hide {} socket
proc socket args {
    puts "New connection: $args"
    tailcall interp invokehidden {} socket {*}$args
}
close [socket www.example.com 80]
→ New connection: www.example.com 80
```

Note the current interpreter is invoking a hidden command within itself. This is possible only because the interpreter is not a safe interpreter.

23.10.3.3. Introspecting hidden commands: interp hidden

```
interp hidden INTERP
CHILD hidden
```

The list of commands that are currently hidden can be retrieved with the `interp hidden` command or the `hidden` subcommand of the interpreter command.

```
% interp hidden safe0
→ socket tcl:file:readlink tcl:zipfs:lmkzip tcl:process:list open pwd tcl:zipfs...
% safe0 hidden
→ socket tcl:file:readlink tcl:zipfs:lmkzip tcl:process:list open pwd tcl:zipfs...
```

In case it is not obvious, the above commands are executed from the safe interpreter's parent, not from within the safe interpreter. Somewhat surprisingly, the safe interpreter can also find out what commands are hidden from it.

```
% safe0 eval {interp hidden {}}
→ socket tcl:file:readlink tcl:zipfs:lmkzip tcl:process:list open pwd tcl:zipfs...
```

23.10.4. Trusting safe interpreters: interp marktrusted

```
interp marktrusted CHILD
CHILD marktrusted
```

The `interp marktrusted` command marks a safe interpreter as trusted. Marking an interpreter as trusted removes **implicit** restrictions.

For example, interpreters created by safe interpreters are automatically safe.

```
interp create -safe ip0 → ip0
interp create {ip0 ip1} → ip0 ip1
interp issafe {ip0 ip1} → 1
```

However, this restriction is removed once the interpreter is marked trusted.

```
ip0 marktrusted         → (empty)
interp create {ip0 ip2} → ip0 ip2
interp issafe {ip0 ip2} → 0
```

However, it does **not** immediately expose any hidden commands. They still have to be explicitly made visible in the child.

```
% ip0 eval {close [open c:/temp/foo.txt w]} ❶
Ø invalid command name "open"
% ip0 expose open
% ip0 eval {close [open c:/temp/foo.txt w]} ❷
```

❶ The `open` command is still hidden ...
❷ ... until it is explicitly exposed

23.10.5. Safe Tcl

As discussed, a completely restricted safe interpreter only has limited use. We therefore described mechanisms that allow execution of potentially dangerous commands within the safe interpreter under the control of the parent. Nevertheless, security is a very tricky business and correct use of these mechanisms without introducing security holes when executing untrusted scripts in a safe interpreter is still fraught with potential pitfalls.

Safe Tcl is a set of commands in the `safe` namespace that simplify the task of extending safe interpreters in a secure manner. It defines aliases for the normally hidden commands `source`, `file`, `encoding`, `load` and `exit`. These aliases expose these commands in the child interpreter but limit their permitted operations as shown in Table 23.1.

Safe Tcl needs to be distinguished from the *safe interpreters* we have discussed above. The former is layered on top of the latter to provide some additional conveniences. The terminology is slightly confusing but follows the Tcl reference documentation.

Table 23.1. Safe Tcl predefined aliases

Alias	Description
`encoding`	Permits all `encoding` (Section 9.1) commands except setting of the system encoding.
`exit`	Deletes the safe child interpreter in which it is invoked instead of terminating the entire process.
`file`	Only allows subcommands `join`, `split`, `dirname`, `extension`, `root`, `tail` and `pathtype`.
`load`	Limits the directories from which extensions can be loaded (Section 23.10.5.5).
`source`	Limits the directories from which scripts can be sourced (Section 23.10.5.5).

23.10.5.1. Creating a Safe Tcl interpreter: safe::interpCreate|interpInit

```
safe::interpCreate ?CHILD? ?OPTIONS?
safe::interpInit CHILD ?OPTIONS?
```

An application can enable the commands in Table 23.1 within a safe interpreter either by creating it with `safe::interpCreate` or adding them to an existing safe interpreter by calling `safe::interpInit`.

The difference with respect to the `interp create -safe` command is that the created interpreters are initialized with the predefined aliases. Let us try out a few commands to illustrate. An interpreter created using `safe::interpCreate` can invoke certain `file` commands that work purely on a syntactic basis while `file` subcommands that access the file system are still not permitted.

```
% safe::interpCreate safeA
→ safeA
% safeA eval {file dirname /var/log}
→ /var
% safeA eval {file isdirectory /var/log}
Ø not allowed to invoke subcommand isdirectory of file
```

In contrast, safe interpreters created using `interp create` cannot invoke the `file` command at all.

```
% interp create -safe safeB
→ safeB
% safeB eval {file dirname /var/log}
Ø invalid command name "file"
```

We can however add the aliases from the Safe Tcl library with `safe::interpInit`.

```
safe::interpInit safeB             → safeB
safeB eval {file dirname /var/log} → /var
```

23.10.5.2. Deleting a Safe Tcl interpreter: safe::interpDelete

```
safe::interpDelete CHILD
```

Interpreters created with `safe::interpCreate` or initialized with `safe::interpInit` must be destroyed with the `safe::interpDelete` command, not with `interp delete` as internal structures related to the added safe extensions need to be cleaned up.

```
safe::interpDelete safeB → (empty)
```

The child interpreter may also terminate itself by calling the `exit` command. Unlike in a normal interpreter, this does not terminate the entire process as it is aliased to the `safe::interpDelete` command.

23.10.5.3. Configuring a Safe Tcl interpreter: safe::interpConfigure

```
safe::interpConfigure CHILD ?options …?
```

Table 23.2 shows options that can be passed to `safe::interpCreate` and `safe::interpInit` to control certain aspects of behaviour. These options can also be set at any later time with the `safe::interpConfigure` command.

Table 23.2. Safe Tcl configuration options

Option	Description
`-accessPath` *DIRECTORIES*	Specifies the list of directories from which the child is allowed to source or load files.
`-autoPath` *DIRECTORIES*	Specifies the list of directories to be included in the child's `auto_path` variable. See Section 23.10.5.5. Not available in Tcl 8.6 and earlier.
`-deleteHook` *SCRIPT*	Specifies a callback script to be run in the **parent** interpreter when the child interpreter is deleted. This script is run **before** the actual deletion and is passed an additional argument which is the name of the child interpreter. An empty string for *SCRIPT* removes any existing callback.
`-nested` *BOOLEAN*	If specified as a boolean `true` value, the child interpreter is permitted to load packages into any interpreters that it creates itself. Default is `false`.
`-nestedLoadOk`	Alias for `-nested true`
`-noStatics`	Alias for `-statics false`.
`-statics` *BOOLEAN*	A Tcl application may include binary extensions that are statically linked into the application itself and thus are not associated with any file or directory paths. If this option is `true` (default), the child interpreter can load such statically linked extensions; if `false`, it cannot.

23.10.5.4. Safe Tcl file paths

```
safe::interpAddToAccessPath CHILD DIRECTORY
safe::interpFindInAccessPath _CHILD _DIRECTORY
```

Even when a safe interpreter needs to be provided access to (some portion of) the file system, it is still considered good practice to not expose the real file system paths to the untrusted scripts. Safe Tcl helps with this by using virtual tokens to represent the physical directories. You can see this by examining an `auto_path` element in the parent and child interpreters.

```
lindex $auto_path 0              → c:/tcl/magic/lib/tcl9.0
safeA eval {lindex $auto_path 0} → $p(:0:)
```

Thus a script running inside the safe interpreter can only see the “virtual” directory tokens and not the real file system paths.

The parent may also add new paths that are permitted to be accessed by the `source` or `load` commands. The `safe::interpAddToAccessPath` command does the needful. The command returns the token that will represent the path in the child.

```
safe::interpAddToAccessPath safeA C:/temp → $p(:235:)
```

Conversely, when passing paths into the Safe Tcl interpreter, the parent needs to translate real paths to the corresponding token. It can use the `safe::interpFindInAccessPath` command for this purpose. For example,

```
safe::interpFindInAccessPath safeA C:/temp → $p(:235:)
```


The Safe Tcl commands in the previous section only provide control of the directories as related to the `source` and `load` commands. They do not help with general purpose file operations. See the `island` package[1] for a solution.

23.10.5.5. Safe Tcl package search

By default, `safe::interpCreate` and `safe::interpInit` set the `auto_path` variable used in package searches to be the same as that of the parent, appropriately tokenized (Section 23.10.5.4). The list of directories from which the child can source or load files is set to list of directories used for auto-loading in the parent. These defaults can be changed by passing the `-autoPath` and `-accessPath` options. Their effect depends on the value of Safe Tcl *sync mode* setting that can be retrieved or modified with the `safe::setSyncMode` command. This setting controls whether `auto_path` is kept in sync with the access path or not. For details, refer to the Tcl `safe` reference documentation.

23.10.5.6. Troubleshooting Safe Tcl interpreters

Debugging scripts in safe interpreters can sometimes be troublesome because certain Tcl features like full stack traces on errors are not made available to avoid information being leaked to the untrusted scripts in the child. Safe Tcl provides the `safe::setLogCmd` command to help with this problem. It permits specification of a script to be run in the parent interpreter when “interesting” events happen in the child. This script is invoked with an additional message describing the event.

```
% safe::setLogCmd puts
% safe::interpCreate safeB
→ NOTICE for child safeB : Created
  NOTICE for child safeB : Setting accessPath=(c:/tcl/magic/lib/tcl9.0 c:/tcl/m...
  NOTICE for child safeB : Setting auto_path=(c:/tcl/magic/lib/tcl9.0 c:/tcl/ma...
  safeB
```

To turn off logging, pass an empty string to the `safe::setLogCmd` command.

[1] https://wiki.tcl-lang.org/page/island

23.11. Setting resource limits

Tcl offers a limited form of protection against runaway scripts in interpreters that result in excessive consumption of computing resources. These limits include

- a maximum recursion depth for execution of Tcl scripts
- a limit on the number of commands that may be executed within an interpreter
- an absolute time by which an interpreter must finish executing a script

We describe these capabilities in this section.

Besides these explicit limits, a parent interpreter may also impose limits using mechanisms already discussed. For example, aliasing `puts` in a child interpreter can limit the number of bytes written to a file by the child.

23.11.1. The recursion limit: interp recursionlimit

```
interp recursionlimit INTERP ?LIMIT?
CHILD recursionlimit ?LIMIT?
```

To protect against runaway scripts overflowing the space allocated for the C stack of the process, Tcl places a limit on the depth of the call stack of an interpreter. An error is raised if this limit is exceeded.

This limit can be retrieved or set with the `interp recursionlimit` command. If *`LIMIT`* is not specified, the command returns the current recursion limit for the specified interpreter (which may be the current one). If *`LIMIT`* is specified, the recursion limit is set to that value.

```
% interp recursionlimit {}
→ 1000
% set ip [interp create]
→ interp4
% $ip recursionlimit 10
→ 10
% $ip eval {
    proc recurse {} {incr ::depth; recurse}
    set depth 0
    catch {recurse} message
    return "Error: failed at depth $depth: $message"
}
→ Error: failed at depth 9: too many nested evaluations (infinite loop?)
```

23.11.2. Limiting number of commands: interp limit

```
interp limit INTERP command ?OPTIONS?
CHILD limit command ?OPTIONS?
```

The command `interp limit command` can be used to limit the number of commands an interpreter is allowed to execute. If no options are specified, the current settings are returned.

```
% interp limit {} command ❶
Ø limits on current interpreter inaccessible
% interp limit $ip command
→ -command {} -granularity 1 -value {}
```

❶ Error because an interpreter may not read its own limits.

The `-granularity` option controls the frequency with which Tcl will check the limit. For example, if set to `5`, Tcl will check the command execution count on every fifth command.

The `-command` option specifies a callback command that will be invoked in the global context of the interpreter that invoked the `interp limit` command when the command count limit is exceeded in the target interpreter. If an empty string, no callback is registered.

The `-value` option specifies the permitted command count. If empty, no limit is placed on the number of commands that may be executed by an interpreter.

Let us play around with changing the limits on an interpreter. We will define a procedure that will be called when the limit is exceeded. A point to note about the call back procedure is that it can **change the limit to permit the interpreter to continue execution**.

```
proc watchdog {ip max} {
    if {[interp limit $ip command -value] < $max} {
        puts "Raising limit to $max"
        interp limit $ip command -value $max
    }
}
```

Now we create a child interpreter and set limits on it. The creation of the interpreter itself executes initialization commands in the child. So when we set limits we may need to take this into account using the `info cmdcount` command to find how many commands have been executed so far.

```
set ip [interp create]
set nexecuted [$ip eval {info cmdcount}]
interp limit $ip command \
    -command [list watchdog $ip [::tcl::mathop::+ $nexecuted 4]] \
    -value [expr {$nexecuted+2}]
```

Now we try executing commands in the child.

```
$ip eval {info cmdcount} → 374
$ip eval {info cmdcount} → 375
$ip eval {info cmdcount} → Raising limit to 377
  376
$ip eval {info cmdcount} → 377
$ip eval {info cmdcount} Ø command count limit exceeded
```

A few words of explanation may be in order.

- We retrieved the command count as of interpreter creation (`nexecuted`).
- We want at most 4 more commands to be permitted to be run and this is the number we pass to `watchdog` as the second argument.
- However, for illustrative purposes we initially set the interpreter command count limit to be only 2 more, not 4.
- Now when we execute commands in the child, the first two execute with not much fanfare. The third one however causes our initial limit to be exceeded and Tcl runs our `watchdog` procedure.
- The `watchdog` procedure **increases** the limit since it has not reached the maximum it was passed.
- After `watchdog` returns, Tcl checks the limit again. Since the **new** limit is greater than the current command count, it permits execution to continue for this command and the next.
- When the new limit is also exceeded, `watchdog` is called again. However, since the limit is now the maximum specified by the `max` argument, it is not increased further.
- Tcl therefore aborts execution of any further commands with an error message.

Note that the interpreter is **not** destroyed when the limit is exceeded. Only errors are generated. The parent interpreter could choose to remove or extend the limits and continue to use the interpreter.

```
$ip limit command -value {} → (empty) ❶
$ip eval {info cmdcount}    → 379
$ip eval {info cmdcount}    → 380 ❷
```

❶ Remove the limit.
❷ Happy days are here again.

23.11.3. Limiting interpreter duration: interp limit

```
interp limit INTERP time ?OPTIONS?
CHILD limit time ?OPTIONS?
```

An alternative way to control the lifetime of an interpreter is by specifying a time by which the interpreter must finish execution. We can retrieve the current settings for an interpreter as for command count limits except that we specify `time` as an argument to `interp limit` instead of `command`.

```
% interp limit $ip time
→ -command {} -granularity 10 -milliseconds {} -seconds {}
```

The `-command` and `-granularity` settings have the same meaning as was discussed in the previous section. The former specifies a callback and the latter controls how frequently limits are checked.

The option `-seconds` is the absolute time, expressed as the number of seconds after the epoch (Section 11.1), beyond which the interpreter must not be allowed to execute. Additionally, the option `-milliseconds` specifies an interval (in milliseconds) beyond the time specified by the `-seconds` value. This is of use in cases where units of seconds is not fine grained enough for the application's purpose. Empty values signify no limits.

Since the general working of time limits is similar to command limits in the previous section (including the ability to change limits in the callback), we only present a simple example.

```
% set now [clock seconds]
→ 1743694541
% $ip limit time -seconds [expr {$now + 2}] -granularity 1
% after 1000 ❶
% $ip eval {set foo 1} ❷
→ 1
% after 1000
% $ip eval {set foo 2} ❸
Ø time limit exceeded
```

❶ Hang around for a second
❷ Succeeds because time limit not crossed
❸ Fails because time limit exceeded

The attempt to set the value of `foo` to `2` fails because the specified absolute time for the interpreter to complete has been exceeded.

We set `-granularity` to 1 above for illustrative purposes as otherwise we would have had to execute more commands to demonstrate the effect. The cost of checking absolute time limits can be significant so in general the granularity should not be lowered for time limits.

23.12. Examples using multiple interpreters

23.12.1. A safe network server

A very common and obvious use of multiple interpreters is to handle client connections in a server by associating each client with a private child interpreter. This has two benefits

- The client contexts are isolated from each other "for free". All context for a client is contained within a child and no special care need be taken to ensure there is no leakage of data or other state between clients.
- Running safe interpreters adds another layer of safety for the server. At the same time, depending on authentication and established authorization roles, different interpreters may be configured with different privileges in terms of what they can execute and access.

Here is a variation of the simple network server we saw in Section 22.1.2. The difference here is that while that server simply reversed the line sent by the client, this server will execute it as an expression, essentially acting as a remote calculator. This is obviously dangerous since the `expr` command can evaluate arbitrary Tcl scripts. We will therefore have to do the evaluation within a safe interpreter.

We start off by defining the command that will invoked in response to every connection request. For each new connection, we create a safe interpreter that will be dedicated to that connection. Furthermore, we place limits on the number of commands that may be executed and the maximum time the interpreter is allowed to live. Then we make the accepted client channel non-blocking and attach a read handler passing it our slave interpreter.

```
proc on_accept {so client_ip client_port} {
    set slave [interp create -safe]
    set cmdlimit [$slave eval {info cmdcount}] ❶
    $slave limit command -value [incr cmdlimit 3]
    $slave limit time -seconds [expr {[clock seconds] + 60}]

    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation \
        crlf
    chan event $so readable [list on_read $slave $so $client_port]
}
```

❶ Get number of commands executed so far in slave

Notice how we set the command limit. The slave internally evaluates commands as part of its internal initialization. So the limit we set has to be some increment to the number of commands already executed by the slave.

Our read handler reads each line from the client and evaluates it as an expression in the slave. For ease of illustration we do not bother dealing with expressions across multiple lines.

```
proc on_read {slave so client_port} {
    set n [gets $so line]
    if {$n >= 0} {
        set status [catch {$slave eval [list expr $line]} result]
        puts $so $result
        if {$status &&
            [lindex $::errorCode 0] eq "TCL" &&
            [lindex $::errorCode 1] eq "LIMIT"} {
            close $so
            interp delete $slave
        }
    } elseif {[chan eof $so]} {
        close $so
        interp delete $slave
    }
}
```

Make a note of the error handling. If the slave evaluation results in an error because of limits being exceeded, we close down the connection. Other errors like invalid input are simply reported back to the client.

Now we start up the server on our local address.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 49999]
vwait forever
```

There is a design choice we made in our implementation to have the master interpreter handle the I/O and invoke the slave only to evaluate the expression. An alternative would have been to hand off the channel to the slave and have it handle the I/O itself. Our choice was based on simpler clean up of the channel when the slave interpreter crossed its limits.

Let us try out our server by connecting to it with telnet.

```
C:\temp\book> start tclsh safe_server.tcl
C:\temp\book> telnet 127.0.0.1 49999
Trying 127.0.0.1...
Connected to 127.0.0.1.
2+2
4
2 ** 4
16
[exit]
invalid command name "exit"
5+5
command count limit exceeded
Connection closed by foreign host.
```

As expected, because of the use of the safe interpreter, the client is not allowed to execute `exit` which would cause the entire server to exit. And once the client has used up its quota of commands, it is kicked out.

You can see that the use of multiple safe interpreters did not add much complexity to the code.

23.12.2. Implementing domain specific languages

Our second example illustrating the use of interpreters involves their use of implementing domain specific languages. A *domain specific language* (DSL) is a programming language that is targeted towards a specific problem domain. The characteristics that distinguish DSL's from general purpose languages like Tcl arise from the fact that users of the language are often not programmers by profession. Rather, they are experts in their respective domains. Consequently, DSL's are generally simple in syntax and programming constructs, often closer in appearance to natural languages. At the same time, their functions are very high level and directly reflect the terminology and abstractions of the target domain.

Since DSL's are not flexible or powerful enough for writing entire applications, they are themselves implemented in the general purpose language, the *hosting* language, used to write the application. An application may support several DSL's at the same time. For example, a database front-end application may include a DSL for queries, a DSL for report generation and a DSL for screen design.

Internal and external DSL's

DSL's may be classified as *internal* or *external*. Internal DSL's are syntactically based on the host language and follow a similar structure. External DSL's on the other hand may follow a completely different syntax from the host language, often because the latter's structure is deemed too foreign from the perspective of the domain experts. Imagine for example if

an accountant were asked to write auditing rules using C++ syntax[2]! The disadvantage of external DSL's is that you now have to write a parser for the DSL syntax which, although not hard for simple DSL's, does involve a non-trivial amount of work.

Happily, with Tcl we can get the benefits of a natural syntax without having to write an external DSL:

- Tcl's syntax as a list of words closely resembles natural language. For example, the Tcl command `deduct federal 10%` is easily understood as English.
- There is no need for declarations, typing and other programming artifacts that an accountant may not be familiar with.
- The DSL can be executed using a separate Tcl interpreter with appropriate safety boundaries.
- Separate interpreters also allows unneeded elements of the language to be easily disabled **if so desired**. This is important because the DSL as seen by the domain expert should be as simple as possible.

For demonstration purposes, we will implement a little internal DSL that an accountant might use to generate paychecks. The following program is a sample that an accountant expects to be run against every employee record in a database to generate paychecks.

```
deduct insurance 100
deduct federal 10% when salary between 20000 and 30000
deduct federal 20% when salary above 30000
deduct state 5% when federal above 2500 ❶
generate paycheck
generate paystub
```

❶ Yes, silly I know but that's taxes for you

Our little language should be understandable to an accountant right away. Most hosting languages would need to implement the above as external DSL's. Not so in Tcl because **it is standard Tcl syntax**.

We start with a couple of utility procedures to check whether a word is included in a permitted list and to ensure a word is an integer. Nothing to see here.

```
proc check_word {word allowed} {
    if {$word ni $allowed} {
        error "Invalid word '$word'. Must be one of [join $allowed ,]"
    }
}
proc check_number {word} {
    if {![string is integer -strict $word]} {
        error "'$word' is not an integer"
    }
}
```

[2] It's hard enough for programmers!

We will assume employee data is passed to us in a dictionary. We will need a procedure to map the keywords in our DSL to keys in the employee dictionary.

```
proc word_to_var {word} {
    dict get {
        salary  Salary
        federal FederalTax
        state   StateTax
        insurance Insurance
    } $word
}
```

As we come to the meat of the implementation, we have a decision to make. The obvious design is to map the commands in the DSL to commands in the safe interpreter that are aliased in the master. At that point, we need to make choice — should the DSL be interpreted or compiled to a Tcl script. In the first case, our alias implementations would be called for every employee record. They would then take the appropriate actions to deduct taxes, print the check and so on. We will go with the second alternative and compile the DSL to a Tcl script. This Tcl script will then be directly evaluated for each employee record. The advantage is that the DSL program need not be repeatedly parsed for every employee record. This method is slightly more complex that the interpreted DSL implementation but since you have already seen metaprogramming in Section 14.4, it should not be hard to follow.

We create a class to store some context — the compiled Tcl script for the DSL code.

```
oo::class create Paymaster {
    variable Script
}
```

Our paymaster needs to be told the rules for generating a paycheck. These rules are provided by our accountant using our DSL. We will pass these in to our paymaster through its constructor.

```
oo::define Paymaster constructor {dslscript} {
    interp create dslengine -safe
    dslengine eval {namespace delete ::} ❶
    foreach alias {deduct generate} {
        dslengine alias $alias [self] $alias
    }
    dslengine eval $dslscript
    interp create calculator -safe
    calculator alias puts puts
}
```

❶ Deleting the global namespace will remove all commands from the slave interpreter.

Our constructor needs some explanation. We create not one, but two, slave interpreters. The first one, `dslengine`, is fundamental to our design because it is the one that will parse the DSL script. We remove **all** commands from it and then add the aliases `deduct` and `generate`

corresponding to the DSL language commands. These then will be the only commands accessible in that slave.

The second interpreter is not strictly necessary. Once `dslengine` has generated the Tcl script, we could run that in the master interpreter. Its presence is to add another layer of security. We do not trust our accountant. He may be annoyed about his annual raise and create a rule that "deducts" `[exec reformat drive]` instead of a number. Although our aliases will validate arguments, there is a chance that something will slip through. Running the generated script in another safe interpreter protects against oversights. Note we cannot use our `dslengine` for this because it does not have any commands other than the DSL-specific ones.

Next we define our destructor which has nothing to do but release the slaves.

```
oo::define Paymaster destructor {
    catch { interp delete dslengine }
    catch { interp delete calculator }
}
```

Time now to define our aliases that implement the DSL. Only two are needed for our simple language. We will start with the more complex one, `deduct`. Remember our design involves **transpiling** the DSL into Tcl so these methods need to generate fragments of Tcl that reflect the DSL statements. These fragments are concatenated to construct the transpiled Tcl script.

We have already seen techniques for constructing scripts in Section 14.3.1.2. We follow that method here by defining a script template that reflects the logic of the script fragment implementing the `deduct` DSL statement. We then construct the script fragments for each place holder in the template and then replace them at the end with `string map`. We will not go into further detail here as that is not the point of this exercise.

```
oo::define Paymaster method deduct {what deduction args} {
    check_word $what {federal state insurance}
    set category [word_to_var $what]
    if {[regexp {^(\d+)(%)?$} $deduction -> amount percent] == 0} {
        error "Invalid deduction amount $deduction"
    }
    set template {
        if {%CONDITION%} {
            if {%USE_PERCENT%} {
                set deduction [expr {int(($Salary * %AMOUNT%)/100)}] ❶
            } else {
                set deduction %AMOUNT% ❷
            }
            incr %CATEGORY% $deduction ❸
            incr NetAmount -$deduction ❹
        }
    }

    if {[llength $args] == 0} {
        set condition true ❺
    } else {
```

```
        if {[llength $args] < 4} {
            error "Incomplete line"
        }
        set args [lassign $args cond_keyword cond_var cond_cmp number]
        check_word $cond_keyword {when if}
        set cond_var [word_to_var $cond_var]
        check_number $number
        switch -exact -- $cond_cmp {
            over - above {
                set condition "\[set $cond_var\] > $number"
            }
            under - below {
                set condition "\[set $cond_var\] < $number"
            }
            within - between {
                if {[llength $args] != 2 ||
                        [lindex $args 0] ne "and"} {
                    error "Invalid \"$cond_cmp\" arguments"
                }
                set upper [lindex $args 1]
                set condition "\[set $cond_var\] > $number && \[set $cond_var\] \
                    < $upper"
            }
        }
    }

    append Script [string map [list \
                                %CATEGORY% $category \
                                %CONDITION% $condition \
                                %AMOUNT% $amount \
                                %USE_PERCENT% [expr {$percent ne ""}]] \
                    $template]
}
```

❶ Deduction is a percentage
❷ Deduction is an absolute value
❸ Allocate deduction to a category...
❹ ...and remove net amount accordingly
❺ No condition for the deduction

The other DSL statement is `generate`. The generation of the script fragment here is much simpler.

```
oo::define Paymaster method generate {what} {
    switch -exact -- $what {
        paycheck {
            append Script {puts "Pay to $Name the amount of $NetAmount"} \n
        }
        paystub {
            append Script {
                puts "Paystub: $Name Salary=$Salary Net=$NetAmount"
                puts "         Fed=$FederalTax State=$StateTax \
                    Insurance=$Insurance"
```

```
            } \n
        }
        default { error "No means of generating \"$what\"" }
    }
}
```

Now we define the method that will be called by the application to pay an employee. The employee record is passed in the form of a dictionary with the employee's name and salary. It then executes the generated Tcl script within the scope of the dictionary in our `calculator` slave.

```
oo::define Paymaster {
    method pay {emp} {
        set emp [dict merge {
            FederalTax 0
            StateTax 0
            Insurance 0
        } $emp]
        dict set emp NetAmount [dict get $emp Salary]
        calculator eval [list set emp $emp]
        calculator eval [list dict with emp $Script]
    }
}
```

Our final method is just for the purposes of inspecting the Tcl script we generated. Just so we can see what havoc we have wrought.

```
oo::define Paymaster method script {} { return $Script }
```

We are ready to try our DSL. Let us create our paymaster, passing in the rules for payment. Normally these would be read in from some resource edited by the accountant.

```
Paymaster create paymaster {
    deduct insurance 100
    deduct federal 10% when salary between 20000 and 30000
    deduct federal 20% when salary above 30000
    deduct state 5% when federal above 2500
    generate paycheck
    generate paystub
}
```

Just for kicks let us see what our generated script looks like.

```
% paymaster script
→
          if {true} {
              if {0} {
                  set deduction [expr {int(($Salary * 100)/100)}]; # Deduction ...
              } else {
...Additional lines omitted...
```

Not pretty, but then again neither are we. At least it seems functional.

```
% paymaster pay {Name Guido Salary 25000}
→ Pay to Guido the amount of 22400
  Paystub: Guido Salary=25000 Net=22400
           Fed=2500 State=0 Insurance=100
% paymaster pay {Name Fredo Salary 35000}
→ Pay to Fredo the amount of 26150
  Paystub: Fredo Salary=35000 Net=26150
           Fed=7000 State=1750 Insurance=100
```

Extending this DSL is fairly easy. You just need to add another aliased method for the new statement, for example `reimburse`.

Let us reiterate what we have accomplished. In essence, we have implemented a domain specific language using syntax that would be natural to a non-programmer. Moreover we have done this without needing to write a parser and in just about a page of code.

And if you are not sufficiently impressed, think about how you would make our DSL **localized** so a French accountant could use the terms he is comfortable with. It would only take a few changes with the help of message catalogs (Section 9.3.2).

That's Tcl.

24

Coroutines

Programs are composed out of computational tasks which may themselves be recursively decomposed into subtasks. There may be dependencies between these tasks where a task may depend on the result of one or more other tasks before it can proceed with its own computation. *Concurrency* refers to the ability of two or more such tasks to progress at the "same time". When we say "same time", we do not necessarily mean that both computations are running at any particular instant. The computations may in fact do that, running in parallel on separate processors, or they may be interleaved on a single processor so that each gets some share of the processor time. In either case, they all progress without any particular task having to be completed first.

Parallelism on the other hand refers to tasks being able to execute **simultaneously** on separate processors so that at any instant they are actually all changing state in some fashion. By definition then, parallelism implies availability of multiple processors each of which is running a separate task.

For a short introduction to the two concepts and their semantic difference, see Rob Pike's talk[1].

We have already seen an example of concurrency in Tcl in our discussion of network server (Chapter 22) implementations. The service provided by a network server to each client can be thought of as a computation task and thus actively servicing multiple clients at the same time through the event loop and asynchronous I/O mechanisms can be thought of as one form of concurrent processing.

We will now look at another form of concurrency using *coroutines*. Like the event loop and asynchronous I/O mechanisms, this allows for concurrent tasks **but not parallel** ones.

At some level, even a procedure can be viewed as a computational task. It is called with some set of input values, performs a computation based on those values and produces a result which may include side effects. Coroutines are similar to procedures in terms of how they are invoked but with one important difference: they run on their own **separate** call stack that maintains the "state" of the computation. Unlike procedures, which lose their **implicit** internal state (local variables etc.) once they return, coroutines can return values (we call this *yielding*) **while maintaining their internal state**. When invoked again, they can then resume from the point where they previously returned.

[1] https://vimeo.com/49718712

We will start off by describing the mechanics of creating and using coroutines. That will allow us to then present their motivation, benefits and working in detail through examples in subsequent sections.

24.1. Creating coroutines: coroutine

```
coroutine CORO CMD ?ARG …?
```

A coroutine is created with the `coroutine` command. It creates a new coroutine context called *CORO* and an associated Tcl command of the same name. The coroutine context consists primarily of a call stack that is used for the execution of all code run in that coroutine's context. The coroutine begins with execution of the command *`CMD`* which is passed any additional arguments provided. This command and any further commands it may call will all run off the newly created coroutine call stack that we will describe later.

Although the coroutine runs with a separate call stack, it still runs in the same interpreter context with the same procedure, global and namespace definitions.

The return value of the coroutine commmand is the value returned by *`CMD`* either when it completes normally, or a value it returns while suspending itself with the `yield` command. Since we have not discussed the latter yet, here is a trivial example of a coroutine that immediately completes normally.

```
proc adder args { return [tcl::mathop::+ {*}$args ] }
coroutine mycoro adder 1 2 3
→ 6
```

This command will create a new coroutine named `mycoro` with its own stack and execute the `adder` procedure within that coroutine context. Our simple procedure computes the sum and immediately returns. The coroutine, along with its associated context, are deleted on the procedure completion. The procedure return value is returned as the result of the `coroutine` command.

All in all, this is just an inefficient way of calling `adder` directly. There is not much point in a coroutine with an independent stack without a means to suspend its execution while preserving the stack context. That's where we will go next.

24.2. Suspending and resuming coroutines

Procedures run to completion returning a result when they are done. As we saw above, coroutines may also behave the same way but their distinguishing ability is being able to suspend their own execution while returning a result to the caller. While suspended, their call stack and context is preserved so that when next invoked, they continue to execute from the point of suspension.

This magic is accomplished with the `yield` and `yieldto` commands.

24.2.1. Yielding to the caller: yield

```
yield ?RESULT?
```

Just like the `return` command in the case of procedures, the `yield` command invoked from a coroutine passes control back to the caller. The difference is that the coroutine call stack context is preserved.

The command may only be invoked from within a coroutine context, i.e. from the command passed to the `coroutine` command as its *`CMD`* argument or from any other command invoked from it to any depth. It preserves the state of the current coroutine context and reverts control back to the coroutine's caller returning *`RESULT`*, which defaults to the empty string, as the result of the coroutine invocation. While the caller continues execution, the coroutine is suspended inside the `yield` command. The coroutine may be called again at any time by its name and passed a single argument at which point it resumes from its suspended state inside of the `yield` command within the coroutine context. The argument passed to the coroutine command is returned as the result of the `yield` command.

From the caller's perspective, the whole sequence of invoking a coroutine looks similar to the following:

```
set result [coroutine CORO CMD ?ARG …?]
set result [CORO ?ARG?]
set result [CORO ?ARG?]
```

The first line above creates and invokes the coroutine. When the coroutine yields, we can invoke it repeatedly using its name. On each such invocation, the corouting resumes its computation from the point of its suspension.

As always, an example will make it all, ahem, crystal clear.

```
coroutine mycoro eval {
    set j [yield 1]
    yield [incr j]
    return 0
}
→ 1
```

The above command creates a coroutine context called `mycoro` and a command of the same name. This coroutine begins executing the `eval` command, **running on a newly allocated stack**. When the first `yield` command is executed, the coroutine execution is suspended and the passed value `1` is returned to the caller of the `coroutine` command. This is what we see as the result above.

Since the coroutine **yielded** back to the caller and did not complete via an implicit or explicit `return`, it remains suspended with its call stack preserved and can be reinvoked with an optional argument.

```
mycoro 10 → 11
```

The above passes control back to the `mycoro` coroutine which resumes execution by returning from the `yield` command. The result of the `yield` command in the coroutine is the argument passed on the coroutine invocation, `10` in this case, which is assigned to variable `j`. The coroutine execution continues with the increment of `j` which is then passed to the second call to `yield` resulting in `mycoro` returning `11` above.

We invoke the coroutine one more time.

```
mycoro → 0
```

Here we did not bother to pass the optional argument to `mycoro` and `yield` therefore would return an empty string as its result. When the coroutine resumes, the next command is a `return`, not a `yield`. The returned value `0` is the result of the `mycoro` call but having completed execution, the coroutine context and call stack are freed and the corresponding command removed. An attempt to call the coroutine will result in an error.

```
% mycoro
Ø invalid command name "mycoro"
```

There are a couple of other points about yielding from coroutines that should be noted. The first is that though our example yields directly from within a simple `eval`, a coroutine can yield from any level in nested procedure calls. Our example could also be mutated as follows:

```
proc demo1 {} {
    set j [yield 1]
    demo2 [incr j]
}
proc demo2 {val} {
    yield $val
}
proc demo {} {
    demo1
    return 0
}
```

We can then define the coroutine as below and run it in the same manner as before.

```
coroutine mycoro demo → 1
mycoro 10             → 11
```

You might expect yielding from a nested procedure call to work and wonder why we spell it out explicitly. This is only because some popular languages that support coroutines only allow yielding from the top level of the coroutine context.

The other point to note is that when a coroutine suspends with the `yield` command, it may only be called with at most one argument. Thus a call like

```
mycoro arg1 arg2
```

will fail. This is not a limitation because (a) multiple arguments can be wrapped and passed as a list, and (b) this capability can be easily added with a wrapper around the `yieldto` command that we will see in Section 24.2.3.

24.2.2. Yielding after initialization

You will often see the main body of a coroutine contain what appears to be an extraneous `yield` before the main section. For example, we want the coroutine `naturals` to return successive integers starting with `1`. We may write it in the following straightforward fashion.

```
proc natural_loop {} {
    while 1 {
        yield [incr i]
    }
}
coroutine naturals natural_loop
→ 1
```

However, the very first `yield` executed in a coroutine is the value of the `coroutine` command itself as seen above. So the first natural number is returned as the result of `coroutine` call and the first `naturals` invocation returns `2`.

```
naturals → 2
```

This is a little awkard and has to be treated specially. Therefore, you will often find coroutines structured such that there is a special `yield` after the coroutine does any internal initialization required. The purpose of this `yield` is simply to return to the caller. The coroutine can then be invoked by its own name for its core functionality.

Our rewritten example would look like

```
proc natural_loop {} {
    set i 0 ❶
    yield ❷
    while 1 {
        yield [incr i] ❸
    }
}
coroutine naturals natural_loop
naturals
→ 1
```

❶ Initialization. Superfluous for our example.
❷ Returns to caller.
❸ Core functionality.

That gives us the desired behaviour. You will find this pattern more often than not in all but the simplest coroutines.

24.2.3. Yielding to arbitrary commands: yieldto

```
yieldto CMD ?ARG …?
```

We earlier described the `tailcall` command (Section 14.1.7) which returns from a procedure. However, unlike the `return` command, `tailcall` does not pass control back to the caller but instead invokes a target command replacing the call frame of the containing procedure invocation with a call to the target command. The result seen by the original caller of the procedure is then the result of the target of the tailcall.

The `yieldto` command is analogous to the `tailcall` command but within the context of coroutines. Like `yield`, it suspends the coroutine in whose context it is invoked but instead of returning control to the caller, it invokes a target command using the same call frame that was used for the coroutine invocation. The result of the original coroutine invocation as seen by the caller is the result of the target command invoked via `yieldto`.

The *`CMD`* and optional *`ARG`* arguments comprise the target command to which control is to be passed. Note that *`CMD`* **may itself be another coroutine**, presenting a way of implementing a "co-operative multitasking" architecture.

Like `yield`, `yieldto` can only be invoked from within a coroutine context and results in that coroutine being suspended. It differs from `yield` in the following respects:

- Whereas `yield` always returns control to the caller of the coroutine, `yieldto` will transfer control to the specified target command.
- When a coroutine suspended using `yield` is resumed, it can be passed at most one argument. On the other hand, a coroutine suspended with `yieldto` can be passed multiple arguments on resumption. The return value from `yieldto` when the coroutine is resumed is the list of arguments that were passed to the coroutine invocation.

The first difference is illustrated by the following example.

```
% coroutine mycoro eval {
    yield ❶
    yield abc ❷
    yieldto string length abc ❸
}
% mycoro
→ abc
% mycoro
→ 3
```

❶ First yield is returned as result of the `coroutine` command itself.
❷ Caller sees `abc` as the return value.
❸ Caller sees result of `string length` as return value.

The second difference with respect to `yield` allows a simple way to construct a `yieldm` command behave like a `yield` in that it returns control to the caller but still allows the coroutine to be resumed with multiple arguments.

```
proc yieldm {{val {}}} {
    tailcall yieldto return -level 0 $val
}
```

(You may wish to revisit Section 15.3 for a refresher on the `return` command's `-level` option.) The implementation of `yieldm` uses `yieldto` to allow the coroutine to be resumed with multiple arguments while passing control to the `return` command to return the specified value to the caller.

As an example of use, consider implementing a coroutine that accumulates the sum of values passed to it and returns the accumulated total on each call.

```
proc accumulate {} {
    set sum 0
    while {1} {
        incr sum [tcl::mathop::+ {*}[yieldm $sum]]
    }
}
coroutine accumulator accumulate
→ 0
```

The use of `yieldm` in lieu of `yield` permits the multiple arguments to be passed to the coroutine in an invocation.

```
accumulator 5     → 5
accumulator 1 2 3 → 11
```

The above illustrated one feature of `yieldto`, the ability to accept multiple arguments on resumption of the coroutine. We will find applications of the other feature, handing off control to arbitrary commands, later in this chapter.

24.3. Checking coroutine context: info coroutine

The `info coroutine` command returns the name of the current coroutine context or an empty string if called from outside a coroutine context.

```
info coroutine             → (empty) ❶
coroutine demo eval {
    yield
    yield [info coroutine]
    yield [info coroutine]
}                          → (empty)
demo                       → ::demo
rename demo demo2          → (empty)
demo2                      → ::demo2 ❷
```

❶ Empty string returned because not within a coroutine context
❷ Renaming a coroutine will be reflected in `info coroutine` as well

The command is generally used in library routines that may change their behaviour based on whether they are running within a coroutine context or not. For example, the `coroutine::auto` package in Tcllib[2] provides a set of I/O commands that use this to hide differences between coroutine and non-coroutine contexts. It implements a version of `read`, `gets` etc. that will behave like the standard `read` outside of a coroutine context. Within coroutines however, it will only block the calling coroutine while yielding to permit other coroutines and non-coroutine scripts to run.

24.4. Coroutine termination

A coroutine and its associated command will continue to exist until it runs to normal completion, raises an uncaught exception, or until the command is deleted with the `rename` command. We will look at exceptions in Section 24.5 so here we will consider just the other two methods.

The first of these is simple — when the command used to create the coroutine completes, the coroutine is also terminated.

```
% coroutine demo eval {yield 0; return}
→ 0
% demo
% demo ❶
Ø invalid command name "demo"
```

❶ Error because the coroutine has completed.

There is a variation of this when a coroutine wants to terminate from within a nested procedure call. This is shown below.

```
% proc exit_coroutine {} {
    return -level [info level]
}
% proc demo_proc {} {
    puts "In demo"
    yield
    exit_coroutine
    puts "Exiting demo"
}
% coroutine demo demo_proc
→ In demo
% demo
% demo
Ø invalid command name "demo"
```

Notice that the `Exiting demo` message is never printed and moreover, the second call to `demo` fails as the coroutine no longer exists. See Section 15.3 if you need a refresher the `return` command's `-level` option.

[2] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

An alternate method of terminating a coroutine is to `rename` (Section 3.5.3) the corresponding command to the empty string. The coroutine may even commit suicide by renaming itself within its own context. In such a case, the coroutine will continue to run until it yields at which point it will be deleted. The following example illustrates this.

```
% coroutine mycoro eval {
    yield
    rename mycoro "" ❶
    puts "Goodbye cruel world!" ❷
    yield ❸
    puts "Exiting coroutine"
}
% mycoro
→ Goodbye cruel world!
% mycoro
Ø invalid command name "mycoro"
```

❶ Deletes the coroutine command.
❷ Coroutine continues to run until the next `yield`
❸ Coroutine destroyed after yielding and cannot be called again.

There is one important difference in the two methods described above. Use of the `return` command can be caught within the coroutine context to prevent termination of the coroutine. This is not possible with the `rename` method which will terminate the coroutine with extreme prejudice after the next yield. Your choice will depend on what's appropriate for your application scenario.

24.4.1. Releasing resources on termination

The fact that termination via the `rename` command will not run any exception handlers has ramifications with regards to resource management. Let us look at the following simple example which returns a line from a file on each invocation.

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    try {
        while {[gets $chan line] >= 0} {
            yield $line
        }
    } finally {
        close $chan
    }
}]
```

The code takes care to ensure that the opened channel is closed on both normal and exceptional completions. However, consider what happens if the coroutine has yielded and the caller deletes the coroutine via `rename`:

```
rename getline ""
```

The `rename` would literally make the coroutine just go Poof! — "disappear" without a trace. In particular, the `finally` handler would never run leaving the file open resulting in leaks.

A robust design warrants that we should guard against this possibility and variable traces provide a way. We can write the coroutine / procedure such that when the `DUMMY` variable goes out of scope, whether it is by the coroutine completing normally or with an exception or **even via the coroutine being deleted as described above**, our trace will trigger and close the file. Note that we do not need to even have a value assigned to `DUMMY` (Section 14.2.1).

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    trace add variable DUMMY unset [list close $chan]
    while {[gets $chan line] >= 0} {
        yield $line
    }
}]
```

24.5. Exception handling in coroutines

Any exceptions that are not caught within a coroutine are propagated to the top level of the coroutine context and passed back to the caller of the coroutine. Moreover, the **coroutine** context and associated command are deleted so that the coroutine cannot be invoked again. We can verify this behaviour by passing our `accumulator` coroutine a non-numeric value.

```
% accumulator notanumber
Ø cannot use non-numeric string "notanumber" as right operand of "+"
% accumulator 3
Ø invalid command name "accumulator"
```

In many cases, it is desirable to not terminate the coroutine but simply notify the caller of the error while preserving the coroutine context so it can be called again. We can accomplish this by combining `yieldto` with one of the exception raising commands. We demonstrated this technique by modifying our `accumulator` implementation as shown below.

```
proc accumulate {} {
    set sum 0
    while {1} {
        if {[catch {
            set part_sum [tcl::mathop::+ {*}[yieldm $sum]]
        } result ropts]} {
            set part_sum \
                [tcl::mathop::+ {*}[yieldto return -options $ropts $result]]
        }
        incr sum $part_sum
    }
}
coroutine accumulator accumulate
→ 0
```

Our modified implementation catches any exceptions raised in the sum operation so that they are not propagated to the top level of the coroutine which is therefore not terminated. At the same time, we need to pass the exception with its original error stack to the caller. We already saw how we can do this for the normal procedural case in Section 15.6.1. Here we cannot use that technique directly because that would again terminate the coroutine. Instead we use `yieldto` to pass the buck. That will suspend the coroutine and run the `return` target command in the context in which the coroutine was invoked. Thus we achieve the goal of returning the exception while preserving the coroutine.

Now when an error occurs, an exception is raised to the caller as before but the coroutine is not terminated and can be invoked again.

```
% accumulator 1 2 3
→ 6
% accumulator notanumber
Ø cannot use non-numeric string "notanumber" as right operand of "+"
% set errorInfo ❶
→ cannot use non-numeric string "notanumber" as right operand of "+"
      while executing
  "tcl::mathop::+ {*}[yieldm $sum]"
      invoked from within
  "accumulator notanumber"
% accumulator 4
→ 10
```

❶ Notice the error stack of the error is preserved as promised

24.6. Variable scopes in coroutines

As we have stated earlier, every new coroutine begins life with its own spanking new stack. The execution however begins in the global context. This is important to keep in mind as the following example shows.

Suppose we had written our `naturals` coroutine example as follows:

```
coroutine naturals eval {
    yield
    while {1} {
        yield [incr i]
    }
}
```

This seems to work as well as the previous version and does not need a separate procedure definition.

```
naturals → 1
naturals → 2
```

The problem is that the code actually modifies the variable `i` in the **global** scope.

```
set i    → 2
naturals → 3
set i    → 3
```

This is undesirable as global variables should be avoided as far as possible. For example, the above implementation would not permit two independent natural number generators to co-exist. Their use of the `i` global would cause mutual interference.

For this reason, it is advisable to have the top level of a coroutine to be invoked as a named or anonymous procedure in all but the simplest cases. Any use of variables then is within the scope of that procedure. We follow this advice in all but the simplest of our examples.

24.6.1. Private variables

The fact that a coroutine runs on its own stack can be taken advantage of to implement "private" variables that are accessible only from code running in that coroutine's context. This is illustrated by the example below from Tcler's Wiki[3]. This defines a procedure `corovars`:

```
proc corovars {args} {
    foreach var $args {
        lappend vars $var $var
    }
    tailcall upvar #1 {*}$vars
}
```

The procedure is simply a wrapper to link local variables in the caller to the local variables of the same name at level 1 in the coroutine stack. It can be used as a "declaration" similar to how the `global` or `variable` commands are used to declare variables in the global and namespace scopes except that it declares variables in the "coroutine" scope.

Here is a demonstration of its use.

```
proc init {} {
    corovars x y
    set x 100
    set y 200
}
proc getx {} {
    corovars x
    yield $x
}
proc demo_helper {} {
    init
    yield
    getx
}
coroutine demo demo_helper
demo
→ 100
```

[3] https://wiki.tcl-lang.org

Notice how the “coroutine-private” variable `x` is referenced from two different procedures running in the coroutine context while remaining inaccessible outside the coroutine.

24.7. Coroutines, uplevel and upvar

A point to be noted about coroutines vis-a-vis procedure calls is that because coroutines run off an independent stack, the `uplevel` and `upvar` commands do not work the same way in the two cases. A procedure may use these commands to run code or access variables in the caller’s context. This is not possible with coroutines. We can see this simply by checking the `info level` command.

```
% coroutine mycoro while 1 {yield [info level]}
→ 0
% proc demo {} {puts "demo: [info level]"; puts "coro: [mycoro]"}
% demo
→ demo: 1
  coro: 0
```

Notice the coroutine is running in the global context and has no access to the stack of the caller `demo`. Of course, all nested procedure calls within a coroutine context can use `upvar` and `uplevel` as normal within that context.

There is however a way to emulate the functionality of `upvar` and `uplevel` with some explicit programming. The idea is to use `yieldto` (Section 24.2.3) to execute a script that does the needful in the caller’s context and then calls back into the coroutine. The technique, accompanied with code and an explanation, is described in TIP 396[4].

24.8. Coroutines and multiple interpreters

Coroutines are limited to the interpreter in which they are defined. For example, the following generates an error.

```
% set ip [interp create]
→ interp0
% coroutine mycoro $ip eval {yield 1}
Ø yield can only be called in a coroutine
```

The error occurs because the coroutine context is defined within the master interpreter but the attempt to yield is made from the slave. If your situation calls for running coroutines within a slave, the coroutine must be defined in that slave. Then you can access it using `interp eval`.

```
% $ip eval { coroutine mycoro try {yield 1; yield 2; yield 3} }
→ 1
% $ip eval mycoro
→ 2
```

[4] https://core.tcl-lang.org/tips/doc/trunk/tip/396.md

Optionally, you can create an alias for some syntactic sugar.

```
interp alias {} mycoro $ip mycoro → mycoro
mycoro                            → 3
```

24.9. Code injection: coroprobe, coroinject

```
coroprobe CORO CMD ?ARG …?
coroinject CORO CMD ?ARG …?
```

Tcl provides two commands that allow code running **outside** a coroutine context to run commands that execute **inside** the context.

The `coroprobe` and `coroinject` commands are not available in Tcl 8.6 and earlier.

The first of these, `coroprobe`, immediately executes a specified command within the context of the targeted coroutine and returns its result. *`CORO`* identifies the coroutine in whose context the command *`CMD`* is to be run. When *`CMD`* runs, it has access to the same context state that existed at the time of the previous yield by that coroutine. Continuing our previous `accumulator` coroutine example, we can use `coroprobe` to reset the `sum` variable within the coroutine to 0.

```
accumulator                     → 10
coroprobe accumulator set sum 0 → 0
accumulator                     → 0
```

At the time the `set` command executes, the `sum` variable is the local variable in the `accumulate` (Section 24.5) procedure that was suspended by the coroutine yielding.

You can use the `coroprobe` command in conjunction with the `info level` command (Section 14.1.2) to dump a coroutine stack and state for debugging purposes.

The second command for running code within a coroutine context is `coroinject`. Unlike `coroprobe`, this does not result in immediate execution of *`CMD`* in the coroutine context. Instead, `coroinject` arranges for *`CMD`* to be executed the next time the coroutine is resumed. At that point, *`CMD`* is run within the coroutine's context **before** the statements following the `yield` or `yieldto` that suspended the coroutine. The arguments passed to the command are the *`ARG`* arguments passed to `coroinject` plus two additional arguments — the first is `yield` or `yieldto` corresponding to the command by which the coroutine had suspended itself, and the second is the value (in the case of `yield`) or list of values (in the case of `yieldto`) passed to the coroutine on its current resumption. The result of the injected command is then passed on as the result of the `yield` or `yieldto` that is returned to the coroutine.

Multiple `coroinject` commands may be invoked before a coroutine is resumed. The injected commands are run in reverse order of their registration with `coroinject` when the coroutine is invoked. The result of each injected command is passed as the resumption value to the next.

Further to our accumulator example, let us write a procedure that can be injected to multiply the passed arguments by a specific multiplier before forwarding them to the coroutine. Note the distinction the code below makes based on the `yield_type` parameter, treating it as a list in the case of `yieldto` and a single value in the case of `yield`. Our accumulator example always used `yieldto` so this was not strictly necessary. In the general case however, where it is not known how the coroutine yielded, this distinction is required.

```
proc injection {multiplier yield_type resume_args} {
    puts "injection: multiplier=$multiplier, yield_type=$yield_type, \
        resume_args=$resume_args"
    if {$yield_type eq "yieldto"} {
        set result [lmap arg $resume_args {
            expr {$multiplier * $arg}
        }]
    } else {
        set result [expr {$multiplier * $resume_args}]
    }
    puts "injection: Passing on resumption arguments $result"
    return $result
}
```

Let us inject our procedure twice.

```
% accumulator
→ 0
% coroinject accumulator injection 2
% coroinject accumulator injection 3
% accumulator 10 20
→ injection: multiplier=3, yield_type=yieldto, resume_args=10 20
  injection: Passing on resumption arguments 30 60
  injection: multiplier=2, yield_type=yieldto, resume_args=30 60
  injection: Passing on resumption arguments 60 120
  180
```

As expected, our injections are executed in reverse order and the coroutine is passed the result of the last execution. Also, the injection does not affect subsequent calls.

```
accumulator 1 2 3 → 186
```

The injected command is free to change the state of the coroutine in any fashion it chooses. For example, it can yield immediately so the actual coroutine code is skipped.

```
proc immediate_yield {args} {yield Skip!} → (empty)
accumulator                               → 186
coroinject accumulator immediate_yield    → (empty)
accumulator 100                           → Skip!
accumulator                               → 186 ❶
```

❶ Note accumulator unchanged; the 100 was not added.

24.10. Using coroutines

Having discussed the mechanics of working with coroutines, we now explore the motivation behind them and their use in some common software patterns. A key point to remember is that coroutines do **not** add any new capabilities to the language. What they do is enable simpler program structure that is easier to write and understand. We will attempt to justify that claim in the next few sections.

24.10.1. Explicit and implicit state

To understand how coroutines can simplify programming, let us first expound a little about the state encapsulated by a computation. This state can be thought of as being a combination of *explicit* and *implicit* states. The former is named such because the program has to explicitly maintain the state as a collection of global, namespace and object variables. The implicit state on the other hand is automatically kept (well, kind of) in the form of the call stack as execution proceeds through a sequence of procedure calls. At any point of time, the explicit and implicit states together tell us where we are in the computation.

Let us illustrate the difference between the two with an example. Consider implementing a task which, given a list, will return the next element in the list on each subsequent call. When the end of the list has been reached, it should wrap back to return elements from the beginning of the list.

Our first implementation will use objects. We could use namespaces as well but that would involve a little more bookkeeping code to deal with the case where we want to use multiple such instances.

```
oo::class create Lgen {
    variable List
    variable Index
    constructor {l} {
        set List $l ❶
        set Index -1
    }
    method next {} {
        incr Index
        if {$Index >= [llength $List]} {
            set Index 0
        }
        return [lindex $List $Index]
    }
}
Lgen create onotes {do re mi}
→ ::onotes
```

❶ We will not bother with error conditions like empty lists

The state of computation is maintained through the member variable `List` and `Index`. Not exactly rocket science because the state information is very simple. We can verify that our implementation meets our requirements.

```
onotes next → do
onotes next → re
```

In contrast, here is a coroutine based implementation that does the same thing through a coroutine.

```
proc list_iterator {lst} {
    yield
    while {1} {
        foreach elem $lst {
            yield $elem
        }
    }
}
coroutine conotes list_iterator {do re me}
```

Just to prove that it works, we try it out.

```
conotes → do
conotes → re
```

The point of our little exercise was to contrast explicit and implicit state. The object based implementation stores explicit state, in the form of the object member variables `List` and `Index`, to keep track of the value to be returned on the next call. On the other hand, in the coroutine-based implementation, the state information is implicitly contained in the `while` and `foreach` loops. This makes the intent of the code more immediately obvious compared to the object-based implementation. This is true even in our trivial example but the difference can be even more pronounced in real world programming tasks with more complex program states.

24.10.2. Generators

One of the simplest uses for coroutines is for implementing *generators*. A generator is a command that returns successive elements from a collection on each invocation. Our `naturals` coroutine and the example in the previous section were essentially generators. In the former case, the "collection" did not actually exist in concrete form (it is infinite after all) but its elements were generated on demand. In the latter case, the generator returned elements of an existing simple (repeated) list.

Here we illustrate a couple of slightly more complex examples of generators, again with the purpose of demonstrating how coroutines simplify their implementation.

Consider a collection of integers stored as a nested list of **arbitrary** depth.

```
% set nested {0 {1 {2 3}} {4 5}}
→ 0 {1 {2 3}} {4 5}
```

Writing a recursive routine to invoke a command on each element of such a tree-like structure is very straightforward.

```
proc iterate {l cmd} {
    set n [llength $l]
    if {$n > 1} {
        foreach e $l {
            iterate $e $cmd
        }
    } elseif {[llength $l] == 1} {
        {*}$cmd [lindex $l 0]
    }
}
```

We can then apply some arbitrary operation to each element, for example output its square.

```
% iterate $nested [lambda i {puts [expr {$i*$i}]}]
→ 0
  1
  4
...Additional lines omitted...
```

However, this is not what we are after. What we actually want is an iterator command that would return the successive integers from this collection on each call.

Generators versus callbacks

The callback model of iteration where a script or command is invoked for every element in a collection is simple and adequate for many uses. However, generators are more flexible in some regards, for instance when the elements are not processed immediately. An example might be if each element was a token to be passed to a client on connecting. Clearly, this would not be suitable for a callback based implementation. Other cases where callbacks are not suitable is when processing of the element depends on significant local state which cannot be passed in the callback although `uplevel` and `upvar` can help some in that regard.

Converting recursive code to an iterator command is normally not possible because by its very nature the recursion encodes the state of the computation and will be lost when the iterator returns. On the other hand, a non-recursive implementation with explicit state involves quite a bit more bookkeeping.

Coroutines come to the rescue. Because they can return values while preserving execution state, we can easily wrap a coroutine around the above to construct an iterator command.

```
proc iterator_wrapper {collection} {
    yield
    iterate $collection yield
}
coroutine iterator iterator_wrapper $nested
```

We can now retrieve one element at a time.

```
iterator → 0
iterator → 1
iterator → 2
```

To better appreciate the simplicity of the above iterator implementation, the reader might want to write one without using coroutines.

Many algorithms, such as tree traversal above, are most clearly expressed in recursive form. In many cases, coroutines can encapsulate these recursive algorithms in a manner that allows them to be used in an iterative manner. Our example was a simple illustration of this.

As another more real world example assume we want an iterator that will return each file or directory within a directory tree. The `fileutil::find` command from Tcllib will return a list of files within a directory.

```
% print_list [fileutil::find c:/temp]
→ c:/temp/fromDir
  c:/temp/newDir
  ...Additional lines omitted...
```

However, we do not want this entire list of files at one shot for many reasons. It may consume too much memory, we may only be interested in the first few entries and so on. We would therefore like an iterator that will return a single entry at a time. Now it so happens that the `fileutil::find` command takes a second optional argument that is the name of a filter command. This command is invoked for each file or directory. If the command returns a boolean true value, the file is included in the returned list. We will misappropriate this feature for our own use.

```
proc yield_one {fname} {
    yield [file join [pwd] $fname]
    return 0
}
proc file_iterator {dir} {
    yield
    fileutil::find $dir yield_one
}
```

We have cunningly fooled `fileutil::find` to yield as part of its filter functionality.

```
% coroutine nextfile file_iterator c:/temp
% nextfile
→ C:/temp/fromDir
% nextfile
→ C:/temp/newDir
% while {[set next [nextfile]] ne ""} { puts $next }
→ C:/temp/fromDir/fileA.txt
  C:/temp/fromDir/subDir
  ...Additional lines omitted...
```

Once again, the benefits of coroutines are best understood by implementing the above example without their use. **The ability to return values from a computation while preserving its implicit state greatly simplifies programming.**

Generators are commonly used in programming and have essentially the same structure in most instances and need to support the same set of high-level operations such as map and fold, something we have not discussed in our earlier examples. Thus it makes sense to implement a framework that provides the common functionality required for generators. The `generator` package in Tcllib does exactly that. The package covers a lot of functionality with many commands and is well documented. You are encouraged to read its source code to enhance your understanding of implementing generators.

24.10.3. Emulating objects

Coroutines can be used to implement simple object based frameworks. Here is the implementation of a simple calculator object.

```
proc plus {args} {
    corovars acc
    set acc [tcl::mathop::+ $acc {*}$args]
}
proc mul {args} {
    corovars acc
    set acc [tcl::mathop::* $acc {*}$args]
}
proc calculator {} {
    set acc 0
    set args [yieldm]
    while 1 {
        if {[llength $args] == 0} {
            set args [yieldm $acc]
        } else {
            switch -exact [lindex $args 0] {
                mul  -
                plus {
                    {*}$args
                    set args [yieldm]
                }
                default {
                    set args [yieldto error "Unknown method [lindex $args 0]"]
                }
            }
        }
    }
}
```

Our implementation could be simpler in the case of our example but we wanted to illustrate the use of private variables using the `corovars` (Section 24.6.1) procedure defined earlier

to serve as member variables. It is even possible to extend this to support prototype-based inheritance with the help of the `yieldto` command.

Proof that our calculator works as designed:

```
% coroutine calc calculator
% calc plus 1 2
% calc mul 5
% calc div 2
Ø Unknown method div
% calc
→ 15
```

Of course our little object system is missing pretty much all the features that Tcl's native OO system provides. But there are situations where a simple object type interface is useful in conjunction with uses of coroutines, for example, the multitasking system we describe in Section 24.10.7.

24.10.4. Producers, consumers, transformers and filters

The producer-consumer pattern is a common pattern in software where the former generates values out of thin air, figuratively speaking, and the latter consumes them for its own internal purposes. Related abstractions are transformers and filters. Both act as consumers, by retrieving values from a producer, as well as producers, by passing those values to a consumer. The difference between the two is that transformers pass on **all** values after potentially modifying them whereas filters may pass on only a subset of them. This has an effect on how they are implemented.

Below we discuss implementation of all these roles in the context of coroutines.

Producers

A coroutine behaves like a producer by returning values to the caller with the `yield` command. Because it is a producer, it does not take any input and in effect ignores the result of `yield` command itself.

We have already seen simple producers, such as the `naturals` example earlier, but it being a legal requirement to include a Fibonacci sequence generator in every discourse on programming, below is a producer that generates such a sequence.

```
proc fibonacci_generate {} {
    yield
    set prev 0
    set fib 1
    while 1 {
        yield $fib ❶
        lassign [list $fib [incr fib $prev]] prev fib
    }
}
coroutine fib_producer fibonacci_generate
```

❶ Note result of `yield` thrown away.

Consumers

A consumer is a parasite, ingesting values without giving anything back in return. Thus a coroutine playing the role of a consumer does not pass any arguments to `yield` but processes its result. Our simple consumer simply prints its argument.

```
coroutine print_consumer while 1 {puts [yield]} ❶
```

❶ Note no arguments passed to `yield`.

Transformers

Transformers have both input and output and therefore pass arguments to `yield` as do producers, as well as use its return values as do consumers. We can write a filter that squares values before passing them on.

```
 coroutine squarer {*}[lambda {} {
     set val [yield]
     while 1 {
         set val [yield [expr {$val*$val}]]
     }
}]
```

We can now hook up the producer and consumer with an intermediate transformer following the pattern

```
CONSUMER [TRANSFORMER [PRODUCER]]
```

In our example then,

```
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 4
```

You can of course extend this chain by adding on additional transformers.

Filters

Filters are a generalized form of transformers in that they remove the restriction of a one-to-one correspondence between input and output values. Most commonly they pass on only a subset of input values (possibly modified) but could potentially insert additional values as well. This generalization means the form we showed above cannot be used for filters. For instance, imagine we had a filter that would only pass through even numbers. We cannot print a list of even fibonacci numbers as

```
print_consumer [evens [fib_producer]]
```

The problem is that if the generated number is not even, the filter cannot pass it to the consumer and nor can it call the producer to retrieve the next number as it is passed values,

not the producer command itself. We have to therefore write filters in a slightly different form where we pass the producer command to the filter when it is created.

Our implementation of the a producer-filter-consumer pipeline that prints even fibonacci numbers would look like

```
proc filter_proc {producer} {
    yield
    while 1 {
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yield $number
        }
    }
}
```

We can then invoke the pipeline as

```
coroutine fib_producer2 fibonacci_generate    → (empty) ❶
coroutine even_fibs filter_proc fib_producer2 → (empty)
print_consumer [even_fibs]                    → 2
print_consumer [even_fibs]                    → 8
```

❶ Fresh copy of our generator

An alternative to the above would be to have the consumer also be driven by the filter as shown below. As a variation, here we also **insert** additional output elements by preceding each the fibonacci number by its position in the **full** Fibonacci sequence.

```
coroutine fib_producer3 fibonacci_generate ❶
coroutine even_fibs2 {*}[lambda {producer consumer} {
    yield
    while 1 {
        incr seq
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yieldto $consumer "Position: $seq"
            yieldto $consumer "Number:   $number"
        }
    }
}] fib_producer3 print_consumer
```

❶ Fresh copy of our generator

Trying it out,

```
even_fibs2 → Position: 3
even_fibs2 → Number:   2
even_fibs2 → Position: 6
even_fibs2 → Number:   8
```

In this last example, we use `yieldto` to call the consumer. As an exercise, consider how this is different from directly calling the consumer coroutine instead of via `yieldto`.

24.10.5. Coroutines and the event loop

Coroutines are invoked just like any other command so there is no great magic involved in calling them via the event loop. However, interaction with the event loop comes up in some common situations, like I/O and multi-tasking, that we describe in later sections so let us go through a simpler example first.

We would like to print the space taken up by each directory in a directory tree. Given that this might take some time on a large file system, we do not want to block the rest of the application in the meanwhile. One way to do this is through a coroutine that reschedules itself through the event loop after doing a block of work. We first define the worker procedure to recurse through the directory tree.

```
proc print_dir_size {dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        incr total [file size $fn]
        if {[file isdirectory $fn]} {
            incr total [print_dir_size $fn]
        }
    }
    puts "$dir: $total"
    after idle after 0 [info coroutine]
    yield
    return $total
}
```

The script is very simple, in part because it ignores errors and special files such as links. After processing each directory, it reschedules itself using the idiom described in Section 19.3.3.1. It then gives up control through `yield`, allowing other parts of the application to run. Note the call to `info coroutine` which returns the name of the current coroutine. The event loop will then use this to resume the coroutine.

If you run this as a coroutine in `wish` or `tclsh` with the event loop running, you will see the output being printed **without blocking the rest of the application**.

```
% coroutine print_windir print_dir_size c:/windows/system32
→ c:/windows/system32/0409: 0
  c:/windows/system32/AdvancedInstallers: 3252576
  c:/windows/system32/AppLocker: 0
  c:/windows/system32/appraiser: 2532564
  c:/windows/system32/ar-SA/Licenses/OEM/Core: 0
  ...Additional lines omitted...
```

It is possible to rewrite the above without the use of coroutines but the resulting solutions are significantly more complex as in every round trip through the event loop you have to explicitly keep track of the current depth in the directory tree and the node within that depth.

We might have given the impression from our examples so far that coroutines are primarily useful in conjunction with recursive algorithms. That is not so. Coroutines are useful any time significant state information is retained in the call stack. Recursive algorithms are just a special category of these and their simple nature makes them particularly suitable for illustrative purposes.

24.10.6. Emulating blocking calls: coroutine::util

As you might have noticed from earlier chapters, particularly those involving I/O, code that uses synchronous blocking calls is both simpler to write and easier to read than its asynchronous counterpart. The drawback of course is that the entire application is blocked if a channel is not ready. Let us look at how coroutines might allow us the benefit of a synchronous style without its drawbacks.

Consider a simple application that acts as a network proxy connected to two remote end points. Its sole purpose in life is to copy data received on a connection to the other connection and vice versa. In real life, such a proxy might have a wide variety of uses such as firewalling, load balancing, protocol translation etc.

The core function of the proxy is simple and ignoring peripheral matters such as connection management, errors etc., it can be implemented by the following procedure.

```
proc proxy {from to} {
    while {[gets $from data] >= 0} {
        puts $to $data
    }
}
```

The procedure copies data from an input channel to an output channel. (There is an implicit assumption here that the two parties are using a text-based protocol.) Our application then is **conceptually** just a pair of calls, one for each direction of the connection.

```
proxy $chan_a $chan_b
proxy $chan_b $chan_a
```

This of course does not work. There are two problems that need to be solved:

- The I/O calls to read data should not block the application.
- The second call needs to run concurrently with the first whereas as written above it will not be invoked until the first completes.

Our coroutine-based version solves both of the above with the help of the `coroutine` package from Tcllib[5]. This package includes several utility procedures that are useful when running in a coroutine context. Once again, this requires the Tcl event loop to be running.

```
package require coroutine → 1.4 ❶
```

❶ Required for coroutine::util::gets

[5] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

We can then write our implementation as follows.

```
proc proxy {from to} {
    while {[coroutine::util gets $from data] >= 0} {
        puts $to $data
    }
}
coroutine atob proxy $chan_a $chan_b
coroutine btoa proxy $chan_b $chan_a
```

Before we get into the explanation of the code, note the similarity of structure with the (non-working) synchronous version we showed earlier. Within the `proxy` procedure we have replaced the `gets` call with a call to `coroutine::util gets`. This takes care of our first problem, blocking. Then instead of calling `proxy` directly, we call it through a coroutine. This takes care of the second problem, concurrency.

The key to the implementation is the `coroutine::util gets` command. This command has the same API as the standard Tcl `gets` command but is intended to be called from a coroutine context so that instead of blocking as `gets` does when no data is available, it invokes `yield` to allow other code to execute while keeping the caller, `proxy` in our case, suspended. It uses the event loop behind the scenes similar to what we described earlier to get called back when data is available. It is well commented and self-explanatory so we will not repeat it here. You can see it in the `coroutine` module of Tcllib or in the online repository[6].

In addition to `gets`, the `coroutine` package includes emulation of several other blocking calls, not all of which deal with I/O. For example, here is a short fragment that keeps writing a message to the console at regular intervals that uses an emulation of the blocking mode of the Tcl `after` command.

```
coroutine print_loop while 1 {
    coroutine::util after 1000; puts "Still going..."
}
```

The Tcl `after` command with a single argument will block until the specified interval has elapsed. The `coroutine::util after` command will block from the coroutine caller's perspective but **will yield under the covers allowing other parts of the application to run**.

24.10.7. Co-operative multitasking

We now move on to our final illustration of coroutine use — a framework for co-operative multitasking or *green threads*. Unlike preemptive multitasking which uses native threads at the operating system level, green threads are implemented purely at the user level and require each thread of computation to voluntarily give up the CPU to allow other threads to run. Hence the term *co-operative* multitasking.

Actors versus Communicating Sequential Processes (CSP)

There are two commonly used models of concurrency — *actors* and *communicating sequential processes (CSP)*. The primary difference between the two is that the former uses

[6] https://core.tcl-lang.org/tcllib/dir?ci=tip&name=modules/coroutine

asynchronous message passing as the communication mechanism. Messages are sent using a “fire and forget” policy where the sender does not wait for the message to be delivered and processed by the receiver. In the CSP model, messages are sent through synchronous *channels* (not to be confused with Tcl I/O channels) where the sender is blocked from proceeding until the receiver accepts the message. Both models can be implemented with Tcl coroutines as evinced by several thirdparty packages.

An example actor implementation

For our illustration of coroutines for co-operative multitasking we will construct a package, `task`, that implements the actor model. Our concurrent units of computation are called tasks. They are implemented as coroutines and voluntarily give up control at appropriate times by yielding to a dispatcher which then gives other tasks a chance to run. Tasks may be completely independent of each other or may interact through *messages*.

Our package is rudimentary and is meant to highlight key considerations involved in implementing a co-operative multitasking framework in Tcl. Amongst other things, a production level package will need to be both more featureful and more robust with better error handling and recovery, resource limits on message queues, a more sophisticated scheduling algorithm to prevent starvations and so on.

We start off by importing the `coroutine` package from Tcllib[7] and defining our package namespace.

```
package require coroutine
namespace eval task {
    variable tasks {}
}
```

Creating a new task

The `task::task` command creates a new task in which the provided script will be evaluated.

```
proc task::task {script} {
    variable next_id
    variable tasks
    variable tasks_added

    set task_id [namespace current]::task#[incr next_id]
    dict set tasks $task_id State INIT
    dict set tasks $task_id Messages {}
    set tasks_added 1
    coroutine $task_id apply {script {
        yield
        try $script finally { task::cleanup [task::myself] }
    }} $script
    wakeup_dispatcher
    return $task_id
}
```

[7] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

The `tasks` variable is a dictionary of active tasks keyed by task identifiers which are dynamically generated using the `next_id` counter. The task identifiers also serve as the name of the coroutine hosting the task. To create a new task, the command initializes the state of the task to `INIT` with an empty `Messages` that will hold any messages sent to it. It then creates a coroutine context in which to run the supplied script. As a general principle, we prefer to run the script within an anonymous procedure so as to avoid potentially polluting the global namespace.

The coroutine immediately yields on entry rather than executing the task script. The reason for this is that we want all execution to be controlled by the dispatcher. Moreover, we do not want the current task to be preempted by the new one unless it voluntarily gives up control explicitly.

Just before returning, we wake up the task dispatcher in case it was previously suspended.

Task identity

A script often needs to know the task under which it is running. The `myself` command returns the id of the current task. If called from outside a task, it raises an exception.

```
proc task::myself   {} {
    variable tasks
    set me [info coroutine]
    if {[dict exists $tasks $me]} {
        return $me
    }
    error "Not running in a task."
}
```

Task cleanup

Any structures allocated by our package need to be cleaned up when the task exits. The `cleanup` command is more or less self explanatory.

```
proc task::cleanup {task_id} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        catch {rename $task_id ""}
        dict unset tasks $task_id
    }
}
```

Note that this does not release any resources allocated by the task itself, for example open channels. That is the responsibility of the task script.

Sending messages

Now we start coming to the meat of the package, implementation of messages and dispatching of tasks. Our package itself does not care about the content of a message. Its interpretation is left entirely to the receiving task. Sending a message involves placing it on the task's message queue and waking up our task dispatcher in case it is not running. By design, we choose to ignore messages sent to non-existing tasks.

```
proc task::send {task_id message} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        dict with tasks $task_id {
            lappend Messages $message
        }
        wakeup_dispatcher
    }
    return
}

proc task::wakeup_dispatcher {} {
    variable dispatcher_alarm
    set dispatcher_alarm 1
}
```

We will explain `wakeup_dispatcher` later when we describe the dispatcher implementation.

Note that as stated earlier message passing is asynchronous in our framework. The sender does not wait for the receiver to accept or process the message.

The message queue

It is worth digressing a little bit at this point to elaborate on a design choice we made to use an explicit message queue as opposed to alternative implementations that you might find being used elsewhere.

The first of these alternatives dispenses with queueing of messages altogether. Sending a message to a task is implemented by calling the corresponding coroutine, either directly or via `yieldto`, passing the message as an argument. You might think of this as a *push* model where messages are pushed to the task whereas ours is a *pull* model where messages are explicitly read by the task with the `receive` command described later. There are two problems with the push method in a general purpose package like ours:

- The task (coroutine) might have yielded for reasons other than for the purpose of receiving messages. For example, the task may use calls such as `coroutine::util gets` or `coroutine::util after` that we saw in previous sections. In such cases, it expects to be resumed from the `gets` I/O handler or from the `after` timer implementation. Being resumed with a message argument at that point instead of the result of a `gets` would completely confuse the poor coroutine.
- Directly pushing messages to coroutines can lead to issues of reentrancy that can be difficult to diagnose and resolve.

The other implementation you might see is the use of Tcl's event queue for passing messages instead of the explicit queue in our design. This also is essentially a *push* model where it is the event dispatcher that is pushing messages to the tasks. This does not have the reentrancy problems but suffers from the first issue noted above.

Maintaining an explicit message queue where tasks pull messages avoids these issues. It also has the added benefit of allowing more sophisticated capabilities such as message priorities, pattern matching etc.

Receiving messages

The complementary command to `send` is `recv` which returns the next message from the tasks message queue.

```
proc task::recv {} {
    variable tasks
    set me [myself]
    while {1} {
        set msgs [dict get $tasks $me Messages]
        if {[llength $msgs] != 0} {
            dict set tasks $me Messages [lassign $msgs msg]
            return $msg
        } else {
            dict set tasks $me State RECEIVE
            suspend
        }
    }
}
```

The implementation is again more or less obvious. If the message queue is not empty, the first message is removed and passed back to the task. Otherwise, the task suspends itself thereby allowing other tasks to run.

Note that in our simple implementation, if a task continuously receives messages, it will not allow other tasks to run. In production quality software, a limit on the number of messages processed before suspension would be well advised. It could be up to the task to do this or our package could be enhanced to keep track of the number of messages received since the last suspension and suspend the thread even if the message queue is not empty.

Relinquishing the CPU

There may be tasks that run independently without sending or receiving messages. Being good citizens however, they may wish to relinquish the CPU to other tasks. The `relinquish` command lets them do that.

```
proc task::relinquish {} {
    variable tasks
    dict set tasks [myself] State RELINQUISHED
    suspend
}
```

We will see the purpose of the state `RELINQUISHED` later.

Suspending a task is nothing more than a simple `yield`.

```
proc task::suspend {} {
    yield
}
```

The task dispatcher

We finally come to the heart of our package, the dispatcher. The role of the dispatcher is to keep looping through the list of tasks looking for any that have messages pending in their receive queue and invoking them. There are several finer points that need discussion but first we show the code.

Our dispatcher is itself a coroutine whose innards are implemented by the `dispatch_loop` procedure. This will loop continuously initializing some state variables and then running an inner loop that iterates over the **current** tasks invoking those that are ready to run. There is a subtle point here in that tasks may be added or removed by any of the current tasks. We will not schedule any new tasks to run until the current tasks have been invoked first. This is a matter of policy; you may choose otherwise.

A task is considered ready to run if

- it is a new task indicated by the `INIT` state.
- it has voluntarily given up the CPU and is awaiting its next turn as indicated by the `RELINQUISHED` state.
- it is waiting for messages as indicated by its `RECEIVE` state **and** there are messages pending in its queue.

```
proc task::dispatch_loop {} {
    variable tasks
    variable dispatcher_alarm
    variable tasks_added

    while {1} {
        set woken 0
        set tasks_added 0
        foreach task_id [dict keys $tasks] {
            if {! [dict exists $tasks $task_id]} continue
            dict with tasks $task_id {} ❶
            if {($State in {INIT RELINQUISHED}) ||
                ($State eq "RECEIVE" && [llength $Messages])} {
                incr woken
                dict set tasks $task_id State RUNNING
                if {[catch { $task_id } mesg]} {
                    puts stderr "$task_id ended with error $mesg"
                    cleanup $task_id
                }
            }
        }
        if {$woken == 0 && $tasks_added == 0} {
            coroutine::util vwait [namespace current]::dispatcher_alarm
        }
    }
}
```

❶ Copy dictionary keys to local variables

We need to make an important point about the task's state — why we mark a task as `INIT`, `RELINQUISHED`, `RECEIVE` or `RUNNING`. This gets back to our earlier discussion of the push versus pull model. A task in the `RUNNING` state may actually have yielded outside of our framework through a call such as `coroutine::util gets`. We must not invoke such tasks from our dispatcher for the same reasons discussed there. Maintaining state information allows us to eliminate these `RUNNING` tasks from consideration. The `RECEIVE` and `RELINQUISHED` states are distinguished by the former indicating the task should be awoken **only** if it has messages waiting, while the latter has no such requirement.

Invocation of a task is done by transferring control to its hosting coroutine. In case of any exceptions, we delete the task from the task database.

Another option for transferring control to the task coroutine would have been to use `yieldto` instead of the direct call. You might find this used in some implementations. We prefer to call the task coroutine directly because our dispatcher would then regain control when the coroutine does a `yield`. This is important for some edge cases where the task yields but not using one of our package commands. For example, a task that sits in a loop using `coroutine::util gets` or the like would relinquish control but not to the dispatcher.

The other detail we need to take care of is with regards to the actions to take when none of the tasks are in a ready condition. Clearly we do not want the dispatcher to just spin endlessly without permitting other code to run. We would like it to suspend itself until there is a reason for it to run. We have a couple of options here. We can pass control to Tcl's event loop after scheduling ourselves to run after some short period of time. Instead of this polling mechanism, we choose the other option: we wait via the `coroutine::util vwait` call for a variable, `dispatcher_alarm` to be set. Correspondingly, the `wakeup_dispatcher` procedure, called on a `send` to any task, sets this variable. If the dispatcher is already active, it has no effect; otherwise, it releases the dispatcher from its suspended state.

Next we have to consider when the dispatcher should suspend itself via the `vwait`. Two conditions should be met for this:

- No tasks should be ready to run. We track this with the `woken` variable which is reset at the top of every dispatcher iteration and incremented every time a task is run. If this is `0` at the end of a complete loop through the task list, we know no tasks are ready.
- No new tasks must have been created since we initiated the iteration through the tasks. Again we use a variable, `tasks_added`, to keep track of this. The variable is reset at the top of the iteration and set when a new task is created within the `task` procedure.

The dispatcher suspends itself when both the above conditions are satisfied.

Starting the dispatcher

We have defined our dispatcher loop procedure but need to provide a means for the application to create it. So we do that now.

```
proc task::start_dispatcher {} {
    coroutine dispatcher dispatch_loop
}
```

Our package is now ready for use.

```
package provide task 1.0
```

We can try out our little package with some simple tasks. The first will simply print any message that is sent to it.

```
set logger [task::task {
    while 1 { puts "Received message: [task::recv]" }
}]
```

The second will send messages to the logger at regular intervals.

```
proc delay100 {} {coroutine::util after 100}
proc now {} {clock format [clock seconds] -format %T}
task::task "while 1 {task::send $logger \[now\]; delay100}"
```

Our final task is a variation of our previous example that printed directory sizes. This version sends the output to the logger task instead of printing it directly and is also considerate to relinquish the CPU.

```
proc dir_size_task {logger dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        if {![catch {file size $fn} size]} {
            incr total $size
        }
        if {[file isdirectory $fn]} {
            if {![catch {dir_size_task $logger $fn} size]} {
                incr total $size
            }
        }
    }
    task::send $logger "$dir: $total"
    task::relinquish
    return $total
}
task::task "dir_size_task $logger c:/windows/system32"
```

We must not forget to start our task dispactcher.

```
task::start_dispatcher
```

Running the above commands in the `wish` shell or `tclsh` **with the event loop active** will result in the console printing the timestamps intermixed with the directory size data.

In our example, we started the dispatcher after creating the task. This is of course not mandated. Tasks can be created after the dispatcher is running as well.

25

ZipFS and Single File Deployment

The Virtual File System, or VFS, abstraction allows applications to view structured data as a hierarchy of directories and files even though it may not stored as such in a real file system. The data can then be operated on with the standard Tcl channel I/O and file commands.

ZipFS is one such VFS that allows the content of ZIP archives to be accessed as a file system from Tcl. As an additional benefit, it also provides utilities to create ZIP archives and *single-file executables* whereby applications and packages can be deployed as a single file without needing additional support files or installation procedures. Commands related to ZipFS are part of the `zipfs` command ensemble.

The `zipfs` command is not available in Tcl 8.6 and earlier.

Tcl's internal VFS framework is also exposed to the script level by the TclVFS extension. It allows for implementation of a VFS purely through scripts, much as the reflected channel API does for channels. See Section A.5.

25.1. Creating ZIP archives

The commands `zipfs mkzip` and `zipfs lmkzip` create ZIP archives from the contents of a directory or a file list respectively.

Let us first create a directory structure for demonstration purposes. We also define a procedure for listing files in an ZIP archive, using commands that we detail later.

```
set dir [file join [pwd] demo]
file mkdir $dir [file join $dir subdir]
writeFile [file join $dir fileA.txt] "This is fileA.txt"
writeFile [file join $dir fileB.txt] "This is fileB.txt"
writeFile [file join $dir subdir fileA.txt] "This is subdir/fileA.txt"
proc zipls {zipfile} {
    zipfs mount $zipfile mnt
    puts [join [lmap path [zipfs find //zipfs:/mnt] {
        string range $path [string length //zipfs:/mnt] end
    }] \n]
    zipfs unmount mnt
}
```

25.1.1. Zipping directories: zipfs mkzip

```
zipfs mkzip ZIPPATH INDIR ?STRIPPATH? ?PASSWORD?
```

The `zipfs mkzip` command creates a ZIP archive from the contents of a directory. *`ZIPPATH`* is the path to the ZIP archive that is to be created. *`INDIR`* is the path to the directory whose contents are to be placed in the archive.

To create a ZIP archive from our demo directory,

```
% puts "Creating ZIP archive from $dir..."
→ Creating ZIP archive from C:/TEMP/book/demo...
% zipfs mkzip demo.zip $dir
```

If we list the content of the archive, we find that **because we passed an absolute source directory path** to `zipfs mkzip`, the archive stores the full path for each added file.

```
% zipls demo.zip
→ /TEMP
  /TEMP/book
  /TEMP/book/demo
  /TEMP/book/demo/fileA.txt
  /TEMP/book/demo/fileB.txt
  /TEMP/book/demo/subdir
  /TEMP/book/demo/subdir/fileA.txt
```

This may be inappropriate as extractors, by default, will recreate the full path on the target system which generally undesired. Passing the optional *`STRIPPATH`* argument will strip the matching prefix from each path. Compare the output below with the one above.

```
% zipfs mkzip demo2.zip $dir [file dirname $dir]/
% zipls demo2.zip
→ /demo
  /demo/fileA.txt
  /demo/fileB.txt
  /demo/subdir
  /demo/subdir/fileA.txt
```

Extractors will then recreate only the `demo` directory as a subdirectory.

The created archive may optionally be encrypted by supplying the *`PASSWORD`* argument to the command.

The encryption algorithm supported by `zipfs` is that from the original ZIP specification and known to be extremely weak. It serves mostly as an obfuscation technique rather than providing real privacy.

25.1.2. Zipping list of files: zipfs lmkzip

```
zipfs lmkzip ZIPPATH FILELIST ?PASSWORD?
```

The second command for creating ZIP archives is `zipfs lmkzip` which creates an archive from a list of file paths. As for `zipfs mkzip`, *`ZIPPATH`* is the path of the created archive and *`PASSWORD`* is an optional password to be used to encrypt the archive. The *`FILELIST`* argument is a list where elements at even indices are paths to files to be included in the archive. The following element is the name to be given to the file within the archive.

```
% zipfs lmkzip demo3.zip [list $dir/fileA.txt first.txt \
                            $dir/subdir/fileA.txt second.txt]
% zipls demo3.zip
→ /first.txt
  /second.txt
```

25.2. The ZipFS file system

Tcl's ZipFS file system only supports a limited number of features of ZIP archives. Amongst the limitations,

- Only the `STORE` and `DEFLATE` storage methods are supported.
- Only weak encryption is supported.
- Multi-part archives and 64 bit formats are not supported.

25.2.1. The ZipFS root

```
zipfs root
```

Before files within a ZIP archive can be accessed using standard Tcl file commands, the archive must be *mounted* as a virtual file system. All ZipFS mounts appear under the *ZipFS root* as returned by the `zipfs root` command.

```
zipfs root → //zipfs:/
```

25.2.2. Mounting ZIP archives as a VFS: zipfs mount

```
zipfs mount ZIPPATH MOUNTPOINT ?PASSWORD?
```

The `zipfs mount` command will mount a ZIP archive as a Tcl VFS file system. *`ZIPPATH`* is the path to the ZIP archive. *`PASSWORD`* is needed when mounting encrypted archives. *`MOUNTPOINT`* is the path under the ZipFS root. The path may or may not be prefixed with the ZipFS root path. In the latter case, the command will implicitly add it. If the empty string, the archive is directly mounted under the ZipFS root. The command returns the full path to the mount point.

```
zipfs mount demo.zip ""                 → //zipfs:/
zipfs mount demo2.zip mnt               → //zipfs:/mnt
zipfs mount demo3.zip //zipfs:/mnt/mnt3 → //zipfs:/mnt/mnt3
```

The mount paths may be nested so `demo3.zip` above is mounted within another mount.

In case of nested mounts, it is advisable to stay away from mounting an archive over an existing file or directory in another mounted archive. At the time of writing, the behavior with respect to restoration of the original file or directory when the nested archive is dismounted is not well defined.

25.2.3. Mounting in-memory ZIP : zipfs mountdata

```
zipfs mountdata ZIPBINARY MOUNTPOINT
```

The `zipfs mountdata` mounts a binary string formatted as a ZIP archive as a ZipFS file system, returning the full path to the mount point. The command is analogous to `zipfs mount` except that the ZIP archive is directly supplied as a binary string instead of a file path.

```
zipfs mountdata [readFile demo.zip binary] //zipfs:/data → //zipfs:/data
```

25.2.4. Introspecting ZipFS mounts: zipfs mount

```
zipfs mount
zipfs mount MOUNTPOINT
```

In addition to mounting ZIP archives, the `zipfs mount` command can also introspect ZipFS mounts using the above syntactic forms. The first of these enumerates all mounted ZipFS archives, returning a dictionary mapping the mount point to the ZIP archive path.

```
% print_dict [zipfs mount]
→ //zipfs:/           = C:/TEMP/book/demo.zip
  //zipfs:/data       = Memory Buffer
  //zipfs:/mnt        = C:/TEMP/book/demo2.zip
  //zipfs:/mnt/mnt3   = C:/TEMP/book/demo3.zip
```

The second form returns the archive mounted at *`MOUNTPOINT`*. The ZipFS prefix is added to *`MOUNTPOINT`* if not already present. If there is no archive at that point, an empty string is returned. If the path is on a different file system, an exception is raised.

```
zipfs mount //zipfs:/            → C:/TEMP/book/demo.zip
zipfs mount //zipfs:/data        → Memory Buffer
zipfs mount //zipfs:/nosuchmount → (empty)
zipfs mount d:/                  Ø Invalid mount path "d:/"
```

25.2.5. Dismounting ZipFS file systems: zipfs unmount

```
zipfs unmount MOUNTPOINT
```

The `zipfs unmount` command dismounts a mounted ZipFS archive. As for `zipfs mount`, *`MOUNTPOINT`* may or may not be prefixed with the ZipFS root.

```
% zipfs unmount //zipfs:/mnt
% zipfs unmount data
% print_dict [zipfs mount]
→ //zipfs:/           = C:/TEMP/book/demo.zip
  //zipfs:/mnt/mnt3   = C:/TEMP/book/demo3.zip
```

See above that although `//zipfs:/mnt` was dismounted, the nested ZipFS mount `//zipfs:/mnt/mnt3` was preserved.

25.2.6. ZipFS utilities: zipfs exists|info|list|find|canonical

```
zipfs canonical ?MOUNTPOINT? FILENAME
zipfs exists PATH
zipfs find DIRECTORY
zipfs info PATH
zipfs list ?-glob|-regexp PATTERN?
```

The `zipfs` ensemble contains several utilities pertaining to ZipFS file systems.

The `zipfs exists` command returns a true Boolean value if the specified path exists within a ZipFS file system and false otherwise.

```
zipfs exists //zipfs:/mnt/mnt3/first.txt → 1
zipfs exists //zipfs:/mnt/mnt3           → 1
zipfs exists fileA.txt                   → 0 ❶
```

❶ //zipfs: prefix missing, not in ZipFS file system.

The `zipfs find` command returns a list of all file paths under a specific directory. The directory need not lie within a ZipFS file system.

```
% zipfs find //zipfs:/mnt
→ //zipfs:/mnt/mnt3 //zipfs:/mnt/mnt3/first.txt //zipfs:/mnt/mnt3/second.txt
% zipfs find [file dirname [info nameofexecutable]]
→ c:/tcl/magic/bin/concrt140.dll c:/tcl/magic/bin/critcl.tclapp c:/tcl/magic/bi...
```

While the `zipfs find` command is not restricted to ZipFS file systems, at the time of writing it does not handle circular links in file systems. This is not an issue on ZipFS file systems as they do not support links.

The `zipfs info` command returns information about a file in a ZipFS file system pertaining to the ZIP archive format. The returned list contains the path of the ZIP archive (as originally passed), the original file size, the compressed file size and the offset of the file's compressed data within the ZIP archive.

```
zipfs info //zipfs:/mnt/mnt3/first.txt → C:/TEMP/book/demo3.zip 17 17 46
```

The `zipfs list` command returns a list of the files in the ZipFS file system irrespective of their mount point. Optionally, a regular expression or glob wildcard pattern may be specified to filter the returned paths.

```
% zipfs list
→ //zipfs:/mnt/mnt3/first.txt //zipfs:/TEMP //zipfs:/mnt/mnt3 //zipfs:/ //zipfs...
% zipfs list -glob *.txt
→ //zipfs:/mnt/mnt3/first.txt //zipfs:/mnt/mnt3/second.txt //zipfs:/TEMP/book/d...
% zipfs list -regexp {/mnt/}
→ //zipfs:/mnt/mnt3/first.txt //zipfs:/mnt/mnt3 //zipfs:/mnt/mnt3/second.txt
```

The `zipfs canonical` command returns the path that a file would be mapped within a mount point. *`MOUNTPOINT`* is prefixed with the ZipFS root if necessary and defaults to the ZipFS root.

```
zipfs canonical /foo/bar         → //zipfs:/foo/bar
zipfs canonical mnt foo          → //zipfs:/mnt/foo
zipfs canonical //zipfs:/mnt foo → //zipfs:/mnt/foo
```

25.2.7. Accessing ZipFS files

Once a ZIP archive is mounted as a ZipFS file system, its contents can be accessed using the standard Tcl commands `file`, `open`, `chan` etc.

```
% set dir [file join [zipfs root] mnt mnt3]
→ //zipfs:/mnt/mnt3
% glob [file join $dir *]
→ //zipfs:/mnt/mnt3/first.txt //zipfs:/mnt/mnt3/second.txt
% set path [file join $dir first.txt]
→ //zipfs:/mnt/mnt3/first.txt
% file stat $path
→ dev 0 ino 0 nlink 0 uid 0 gid 0 size 17 atime 1743694542 mtime 1743694542 cti...
% readFile $path
→ This is fileA.txt
% writeFile $path "New text"
```

While existing files can be written to as above, directory operations such as creation and deletion of files and subdirectories are not supported as of Tcl 9.0.

Any changes made to a ZipFS file content are only made in memory. They are not persisted to the ZIP archive and are lost when the file system is dismounted.

25.3. Single file deployment

Except in the simplest cases, deploying an application or even a large library package involves distribution of

- The Tcl interpreter itself, e.g. `tclsh` or a custom built executable
- The Tcl script files comprising the application
- Any packages, modules and binary extensions required
- Support files such as icons and other resources

The presence of multiple files in a directory hierarchy means deployment cannot be a simple copy and run operation. The files have to combined into some archive or installation format for distribution and then unpacked or installed on the target system. Although this can be an inconvenience, there is another more irksome issue for the user. If multiple unrelated Tcl applications are installed, there is potential for interference between applications. Each may require different versions of Tcl and libraries, expect different settings in environment variables like `TCLLIBPATH` and so on.

The *single-file executable (SFE)* offers a solution to the above. It comprises of a static build of `tclsh` (or `wish` or a custom executable) to which a ZIP archive, containing any required scripts, packages and binary extensions, is attached. At runtime, the Tcl interpreter loads packages, extensions and other support files from this archive making the executable a self-contained application that can be deployed as a single file and run without any additional installation steps.

Before we delve into how these single file executables can be created, we need to first look at embedded ZIP archives.

25.3.1. Embedded ZIP archives

Earlier sections in this chapter described the mounting of ZIP archives as file systems within a Tcl interpreter. What makes this facility even more useful is that **the ZIP archive can even be one that is appended to either the application executable or the Tcl interpreter shared library**. We briefly mentioned in Section 2.1.3.2 how the Tcl support scripts can be attached as a ZIP archive to the executable or shared library. We now go into some detail how this is effected.

- On platforms that support this facility, including Windows, Linux and MacOS, a ZIP archive can be appended to an executable binary without impacting the operating system's recognition of the file as executable.
- In the case of a Tcl build configured to generated a shared library, when the `tclsh` shell initializes, it examines if the shared library has such an archive appended. If so, it is mounted as a ZipFS file system at `//zipfs:/lib/tcl`. Further, if the archive has a top level directory `tcl_library` containing the file `init.tcl`, it is assumed to contain the Tcl support scripts and the `tcl_library` global variable (Section 17.1) is set to point to it.
- Similarly, if a ZIP archive is found attached to the main executable `tclsh`, it is mounted as a ZipFS file system at `//zipfs:/app`. If `tclsh` was linked statically[1]. and the archive has

[1] Implies there is no shared library containing an attached `tcl_library` archive

a toplevel directory `tcl_library` containing `init.tcl`, the `tcl_library` global is set to point to it.

- Finally, if the archive attached to the `tclsh` executable contains the file `main.tcl` at the top level, `tclsh` sources it as the main application script.

Below is the output from a `tclsh` built with the default shared library configuration with an embedded ZIP.

```
% set tcl_library
→ //zipfs:/lib/tcl/tcl_library
% zipfs mount //zipfs:/lib/tcl
→ C:/Tcl/9.0.0/x64/bin/tcl90.dll
```

In the case of the `wish` shell, the Tk support scripts are handled similarly except the archive containing them is attached to the Tk shared library and mounted at `//zipfs:/lib/tk`.

```
% puts $tk_library
→ //zipfs:/lib/tk/tk_library
% zipfs mount
→ //zipfs:/lib/tk C:/Tcl/9.0.0/x64/bin/tcl9tk90.dll //zipfs:/lib/tcl C:/Tcl/9.0...
```

25.3.2. Single-file applications

```
zipfs mkimg OUTFILE INDIR ?STRIPPATH? ?PASSWORD? ?IMAGEPATH?
zipfs lmkimg OUTFILE FILELIST ?PASSWORD? ?IMAGEPATH?
```

Based on the above discussion, building a single file application therefore involves the following steps:

- Build or download a statically built `tclsh` (or `wish` for Tk). A shared library build is obviously not usable as the main executable and Tcl library are separate components.
- Create a directory containing the Tcl, and optionally Tk, support scripts along with the application code. The top level of the directory should have a `main.tcl` file that loads and initializes the application.
- Create a ZIP archive of the directory and attach it to the static executable with the `zipfs mkimg` or `zipfs lmkimg` commands.

The *`INDIR`*, *`FILELIST`*, *`STRIPPATH`* and *`PASSWORD`* parameters are used to create the ZIP archive. These are analogous to the parameters for the `zipfs mkzip` (Section 25.1.1) and `zip lmkzip` (Section 25.1.2) commands described earlier. The commands attach the created archive to the contents of the file specified by *`IMAGEPATH`*, storing the combined content in the file *`OUTFILE`*. If *`IMAGEPATH`* is not specified, it defaults to the path returned by `info nameofexecutable`, which is normally the path of the currently executing process.

If the inputs to the archive (*`INDIR`* or *`FILELIST`*) are defined appropriately, and *`IMAGEPATH`* is a statically bound Tcl application, the created *`OUTFILE`* will be a standalone, self-contained application.

For illustrative purposes, let us create a standalone application that can be copied and run from anywhere.

```
% file mkdir voila.vfs ❶
% file copy $tcl_library [file join voila.vfs tcl_library] ❷
% writeFile [file join voila.vfs main.tcl] {puts "Voila!"} ❸
% zipfs mkimg voila.exe voila.vfs voila.vfs "" \
    C:/Tcl/9.0.1/x64-static/bin/tclsh90s.exe ❹
% exec voila.exe
→ Voila!
```

❶ Make a directory to contain our application
❷ Copy the Tcl support scripts
❸ Write our main application script
❹ Generate a ZIP archive and attach it to the static Tcl application

Two points to be noted above:

- The *`STRIPPATH`* argument is passed as the name of the source directory being wrapped in to the archive. Otherwise, there will be `hello.vfs` directory at the root of the archive with the actual contents underneath it.
- The last argument is a path to a static build of `tclsh`. If the above commands are run from within a static `tclsh` the last argument can be omitted.

Creating a `wish` based Tk GUI application follows a similar path with minor changes. In addition to copying the contents of the `tcl_library` directory, the contents of `tk_library` also need to be copied to the source directory. The *`IMAGEPATH`* argument should be the location of a statically bound `wish` executable and not `tclsh`.

In a real world application, there will be a number of support packages, additional application scripts and so forth. These can be placed within the directory being wrapped. It is the responsibility of the `main.tcl` script then to set up the `auto_path` variable for packages, source application scripts etc. as appropriate. If any packages include binary extensions, the package loading commands will automatically take care of copying the binary extensions to a native file system so that they can be loaded by the system. However, any resource that will be accessed from other applications or the system must be copied to the native file system. This includes any support shared libraries that are not directly loaded by Tcl.

26

Testing and Performance

> Quality is not an act, it is a habit.
>
> — Aristotle

A very large part of every software development process is (or should be!) ensuring quality — making sure that the software works as expected and at a satisfactory performance level. This chapter describes some of the tools available within Tcl for meeting these goals in Tcl applications.

26.1. Testing

If developing software in Tcl, the interactive nature of Tcl streamlines the development of unit tests. As you code procedures and commands, you can, and **must**, interactively call them from a Tcl shell to verify they work correctly with various inputs. On failures, you can fix the code and reload the changes into the shell with the `source` command for further testing. This mode of development should, as the great philosopher stated, become a habit.

These interactive "off-the-cuff" tests can then be easily formalized, expanded and translated to a form suitable for inclusion in an automated suite. A comprehensive test suite is crucial to shipping robust software. The direct benefits are not just a better customer experience and reduced support costs but also freedom to refactor, optimize and enhance the software while maintaining a high level of confidence that working functionality is not broken by the changes.

Even when the software under test is written in a different language, by its very nature Tcl is very well suited as a language for building test automation suites and is widely used for that purpose. Several commercial test automation products as well as the open source frameworks such as DejaGNU[1] and the Selenium based Caius[2] framework are based on Tcl.

Here we will only describe the `tcltest` package which comes with the Tcl source distribution and can be used for testing your own packages as well as applications.

26.1.1. The tcltest package

The `tcltest` package is included in the Tcl source distribution and used in the testing of not only Tcl itself but many other packages and extensions. The package is included in many but not all binary distributions. You can check if it is installed by attempting to load it.

[1] http://www.gnu.org/software/dejagnu/
[2] http://caiusproject.com

```
package require tcltest → 2.5.9
```

If you get an error, it may need to be separately installed with your system package manager or extracted from a source distribution into a directory included in your Tcl installation's `auto_path` (Section 17.2) variable.

A `tcltest`-based test script consists primarily of a series of `tcltest::test` commands.

```
tcltest::test NAME DESCRIPTION ?OPTION …?
```

The *`NAME`* argument is a label given to the test for identification purposes. It is recommended to use names with some structure so as to be able select a subset of tests using glob wildcard matching. The *`DESCRIPTION`* argument is strictly for human consumption but clear descriptions greatly help in formulating a complete test suite. The rest of the arguments are specified as options and we will describe these as we go through our example.

We will demonstrate the use of the package with a simple example, a script containing some tests for Tcl's `incr` command. Test scripts using the package are simply Tcl scripts so you can use any Tcl commands, load other packages and so on. By convention, test scripts are given the `.test` extension so we name our test script `incr.test`.

The test script should start off by loading the `tcltest` package. You may also load any other packages required for testing, define supporting procedures and so on. For our simple example, we do not need anything else.

```
package require tcltest
```

A test case is defined with the `tcltest::test` command. For convenience we import this command to save us some typing.

```
namespace import tcltest::test
```

Let us write a test for the basic use of `incr`.

```
test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}
```

The first argument labels the test case. The second describes the test. The `-setup` option specifies a script that does any required set up before the actual test can be run. In our case, it just initializes a variable to a known value. The `-body` option specifies the actual test itself. For our test case, it is meant to check the result of the command as well the new value of the variable. Many times the `-setup` script is combined with the `-body` script but we

do not recommend this as it obfuscates exactly what is being tested. The `-cleanup` option specifies a script to run to remove any side-effects of the test case. Finally, the `-result` option specifies the value that is expected as the result of executing the script provided as the `-body` argument.

The options may be specified in any oder and not all need be specified though obviously it does not make sense to have a test case without a `-body` option. The `-result` option will default to the empty string.

It is important for a test suite not only to verify correct behaviour on valid input, but to also verify that the command handles bad input correctly. So we will now define a test case to ensure invalid input is correctly rejected by the command.

```
test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp
```

There are two additional options we see here. The `-returnCodes` option specifies the expected return code from evaluation of the `-body` script. By default, as in our previous test case, it is expected to be `ok` indicating a normal script completion. Here we expect an error exception and accordingly specify that as the `-returnCodes` value. The `-result` option now holds the expected error message. Since error messages may change slightly between builds, we do not want to specify an exact message value. We therefore use the `-match` option to indicate that the `-result` option value should be treated as a regular expression to be matched with the `-body` script result. You may also specify glob matching or even define your own matching criteria.

Having compiled a comprehensive set of tests, we end our test script with a call to clean up.

```
::tcltest::cleanupTests
```

The `tcltest::cleanupTests` command does two things. It cleans up any temporary files and other changes made by the test harness. It also prints out summary statistics from the test run.

Here is our entire test script.

```
# incr.test
package require tcltest

namespace import tcltest::test

test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
```

```
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}

test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp

::tcltest::cleanupTests
```

Let us try running this test script from the Windows or Unix shell.

```
C:\temp>tclsh incr.test
incr.test:      Total   2       Passed  2       Skipped 0       Failed  0
```

In case of any failures, `tcltest` will print details about the failing test cases.

We can also choose to run a subset of the tests by matching test labels and ask for more verbose output.

```
C:\temp>tclsh incr.test -match *err* -verbose p
++++ incr-err-1.0 PASSED
incr.test:      Total   2       Passed  1       Skipped 1       Failed  0
```

Notice only the error test case was run. There is also a `-skip` option that is complementary to `-match` and specifies tests that should be skipped.

There are a number of other features that we will not describe here. These include, among others

- ability to set up test constraints so that tests are only run if certain conditions are satisfied, for example limiting certain tests to specific platforms
- checking expected output on standard channels
- utility commands for creation and clean up of temporary files and directories.

Hai Vu's blog[3] contains series of posts related to both basic and advanced use of the `tcltest` package. You may find it a more accessible tutorial for the `tcltest` package than the reference documentation.

[3] https://wuhrr.wordpress.com/category/programming/tcl-programming/

26.2. Improving performance

> If you optimize everything, you will always be unhappy.
>
> — Donald Knuth

Given that, improving performance involves multiple steps:

- profiling the application to determine which parts of the application are the "hotspots" that need attention
- measuring execution time for implementations of alternative algorithms for these hotspots
- **if needed**, examining the code for possible "microoptimizations" that can nevertheless significantly speed up the program under certain circumstances.

The next few sections go into these topics.

26.2.1. Profiling scripts

Writing a basic profiler for Tcl scripts is easy because of Tcl's malleability and introspective capabilities. For example, procedures can be redefined or wrapped to collect timing information on entry and exit. An alternative method is to use the command and execution tracing facility described in Section 14.2. You will find several such profilers referenced in Tcler's Wiki[4]. Here we will describe the `profiler` package included in Tcllib[5].

Note however that although writing the profiler is easy, interpreting the data needs to be done with care for many reasons. The statistics relating to call counts are generally accurate but timing information is less so, particularly for profilers measuring at the source line level. The profiler itself has significant impact on speed of execution. It is best to not rely on profiler output for fine grained measurements and use the output for **relative** comparison of where time is being spent.

When profiling an application, the `profiler` package must be loaded and the `profiler::init` command called **before** any of the procedures to be profiled are defined.

```
package require profiler → 0.7
profiler::init           → (empty)
```

Let us now write some sample code that we will use for demonstration purposes. The procedures do not do anything but pretend they are doing useful work by napping.

```
proc fiddle {} { after 10 ; twiddle }
proc twiddle {} { after 20 }
proc diddle {} { after 30 }
proc main {} {
    for {set i 0} {$i < 10} {incr i} {
        fiddle
        diddle
    }
}
```

[4] https://wiki.tcl-lang.org
[5] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

Now we call `main` and then print the collected information.

```
% main
% profiler::print ::fiddle
→ Profiling information for ::fiddle
  ============================================================
              Total calls:  10
      Caller distribution:
    ::main:  10
             Compile time:  597602
            Total runtime:  5125132
          Average runtime:  512513
            Runtime StDev:  88460
           Runtime cov(%):  17.3
    Total descendant time:  3169993
  Average descendant time:  316999
  Descendants:
    ::twiddle:  10
```

We have only printed the profiler data for `fiddle`. If we had not provided an argument to `profiler::print` it would have printed it for all procedures.

The `profiler::print` command requires the **fully qualified** command name to be provided.

Most of the information is self-explanatory. The one item that needs some explanation is the `Compile time` line. The first time a procedure is invoked, Tcl compiles its body into bytecode. Thus the first invocation of a procedure is often significantly longer than subsequent invocation. This difference does not show up in our example because our artificial delay swamps any additional compilation time required.

An alternative to `profiler::print` is the `profiler::dump` command which returns the same information in a dictionary format. The package offers utility commands to suspend, resume and reset profiling specific procedures or all of them.

26.2.2. Timing scripts

Having pinpointed what parts of the application need to be worked on, we can try different algorithms to improve performance, say using a skip list versus a binary search tree. We then need a means for measuring execution time for each alternative. Tcl provides two commands for this purpose, `timerate` and `time`. We will mostly focus on the newer `timerate` command as it is both more accurate and more flexible.

26.2.3. The timerate command

```
timerate ?options? SCRIPT ?INTERVAL? ?MAXCOUNT?
```

The `timerate` command is available in Tcl 9 and newer patch releases of Tcl 8.6. The command runs the specified script multiple times, returning the average elapsed time for a

run. *INTERVAL* specifies how long the measurement should run and defaults to 1000 ms. The duration of the measurement can be further restricted by *MAXCOUNT* which is the maximum number of iterations of *SCRIPT*.

Let us demonstrate the use of `timerate` by comparing timings for the braced and unbraced variations of the `expr` (Section 7.2.2) command.

Before the actual measurement, calibration is required to account for any overhead in the measurement process itself. The `-calibrate` option is used for this purpose.

```
% timerate -calibrate {}
→ 0.039028199091523434 µs/#-overhead 0.040559 µs/# 48126987 # 24655218 #/sec
```

The measured value is stored internally by `timerate` as the default measurement overhead to be assumed in further invocations of the command.

Once calibrated, we can then run the timing tests for comparison.

```
% set i 1
→ 1
% timerate {expr $i+$i} ❶
→ 1.107789 µs/# 871979 # 902698 #/sec 965.969 net-ms
% timerate {expr {$i+$i}} ❷
→ 0.125295 µs/# 6085586 # 7981190 #/sec 762.491 net-ms
```

❶ Unbraced syntax
❷ Braced syntax

The result returned by the command is a well formed list containing the average time per iteration in microseconds, the number of iterations that were run, the estimated number of executions per second, and estimated total execution time excluding measurement overhead.

As seen from the results above, brace[6] your `expr` expressions.

The script passed for calibration should generally be the empty script. A non-empty script can be passed if it is also included in the later scripts being measured where it should be considered overhead and not contribute to the measurements. For example, suppose we want to measure the performance of `lappend` for a list of a thousand elements. The cost of creation of the list should not be included in our timing. We can calibrate accordingly before measurement.

```
% timerate -calibrate {set L [lrepeat 1000 X]} ❶
→ 2.2022767113875212 µs/#-overhead 2.220777 µs/# 878973 # 450292 #/sec
% timerate {set L [lrepeat 1000 X]; lappend L Y} ❷
→ 0.332167 µs/# 394564 # 3010537 #/sec 131.061 net-ms
```

❶ Measure overhead to be ignored.
❷ Actual measurement of `lappend`.

[6] Performance is only one reason to brace your expressions. See Section 7.2.2.

The overhead measured with `-calibrate` is used for all further measurements until the next calibration. Remember to reset it as appropriate.

As an alternative to changing the calibration, `timerate` accepts the `-overhead` option that explicitly specifies the overhead to be ignored.

```
% set overhead [lindex [timerate -overhead 0 {set L [lrepeat 1000 X]}] 0] ❶
→ 2.211364
% timerate -overhead $overhead {set L [lrepeat 1000 X]; lappend L Y} ❷
→ 0.313080 µs/# 396129 # 3194073 #/sec 124.020 net-ms
```

❶ Measure overhead of lrepeat.
❷ Actual measurement of `lappend` explicitly ignoring overhead.

This has the advantage of not having to reset the calibration for the next measurement.

The value passed with `-overhead` is treated as the total overhead **including the intrinsic overhead** of the measurement process set by the `-calibrate` option. Therefore, when measuring the overhead itself using `timerate`, the value of `-overhead` should be passed as `0` as in the example above.

One final option accepted by `timerate` is `-direct`. By default, `timerate` compiles the passed script into bytecode before measuring performance. If the `-direct` option is specified, the script is executed without compilation into bytecode.

```
% timerate -calibrate {}
→ 0.03982422701274929 µs/#-overhead 0.040174 µs/# 48588339 # 24891567 #/sec
% timerate {lindex $L 0}
→ 0.062694 µs/# 9754413 # 15950598 #/sec 611.539 net-ms
% timerate -direct {lindex $L 0}
→ 0.108374 µs/# 6747734 # 9227329 #/sec 731.277 net-ms
```

26.2.4. The time command

```
time SCRIPT ?COUNT?
```

The `time` command is a less accurate alternative to `timerate` available in older versions of Tcl. It evaluates *`SCRIPT`* a *`COUNT`* number (default once) of times and returns a string describing average execution time. A sufficiently large *`COUNT`* value should be passed because the first execution of a script will often result in the script being compiled to bytecode resulting in an additional cost. Unlike `timerate`, `time` does not compensate for measurement overhead.

```
% time {lindex $L 0} 10000
→ 0.13062 microseconds per iteration
```

Appendix A. Core packages

A few packages, while not part of the core language, are shipped as part of the core distribution, either because they are widely used or for historical reasons.

The `tdbc`, `thread` and `vfs` packages were covered in the first edition of this book but have been removed from the second edition for reasons of space. The related chapters from the first edition can be downloaded from the book website[1].

A.1. The Tk graphical toolkit

The most well known Tcl extension is the `Tk` graphical toolkit. Its portability, widget set and ease of use has made it the de facto graphical toolkit of choice not just for Tcl but also for other languages. The online book http://www.tkdocs.com covers modern Tk usage.

A.2. The tdbc extension

The *Tcl Database Connectivity* (TDBC) extension provides a standard Tcl API for accessing SQL databases from multiple vendors.

A.3. The http and cookiejar packages

The `http` and `cookiejar` packages implement the client-side HTTP protocol with cookie support. We touched on this very briefly in (Section 22.1.4). See the Tcl reference documentation for all its capabilities.

A.4. The Thread extension

The `Thread` package exposes Tcl's multithreading capabilities to the scripting level.

A.5. The tclvfs extension

The `vfs` extension allows for Tcl virtual file systems to be implemented in script. It additionally includes VFS wrappers for various archive formats such as tar as well as network protocols like WebDAV.

A.6. The registry extension

The `registry` package implements a simple interface to access the registry on Windows platforms. See the `registry` page in the Tcl reference documentation for more information.

A.7. The dde extension

The `dde` package implements the Dynamic Data Exchange (DDE) protocol on Windows platforms. DDE is generally considered obsolete technology and should only be used for legacy applications. See the `dde` page in the Tcl reference documentation.

[1] https://www.magicsplat.com/ttpl/index.html

A.8. The IncrTcl extension

IncrTcl was one of the early object-oriented languages layered on Tcl. It is available as the `Itcl` package for legacy applications that rely on it.

Appendix B. Utility scripts

We use some simple utility scripts in this book. These are listed here.

The `print_args` utility simply prints the arguments passed to it, separated by commas.

```
proc print_args {args} {
    puts "Args: [join $args {, }]"
}
```

The `print_list` utility prints each element of a list on a new line. The `print_sorted` utility is similar but prints them in sorted order.

```
proc print_list {l} {
    puts [join $l \n]
}
proc print_sorted {l} {
    print_list [lsort -dictionary $l]
}
```

The `print_dict` utility, from the Tcllib[1] `debug` module prints a formatted dictionary.

```
proc print_dict {dict args} {
    if {[llength $args] == 0} {
        set names [lsort -dict [dict keys $dict]]
    } else {
        set names {}
        foreach pattern $args {

            lappend names {*}[lsort -dict [dict keys $dict $pattern]]
        }
    }
    set maxl 0
    foreach name $names {
        if {[string length $name] > $maxl} {
            set maxl [string length $name]
        }
    }
    set maxl [expr {$maxl + 2}]
    set lines {}
    foreach name $names {
        set nameString [format %s $name]
        lappend lines [format "%-*s = %s"  $maxl $nameString  [dict get $dict \
            $name]]
    }
    puts [join $lines \n]
}
```

[1] https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/toc.md

The `print_array` prints the contents of an array or a subset thereof.

```
proc print_array {args} {
    uplevel 1 parray $args
}
```

The `wait` utility enters the event loop for the specified amount of time.

```
proc wait {ms} {
    after $ms [list set ::_wait_flag 1]
    vwait ::_wait_flag
}
```

The `lambda` command is syntactic sugar for defining an anonymous procedure.

```
proc lambda {params body args} {
    return [list ::apply [list $params $body] {*}$args]
}
```

The `bin2hex` command pretty prints binary data in hex.

```
proc bin2hex {args} {
    regexp -inline -all .. [binary encode hex [join $args ""]]
}
```

Index

Symbols

::tcl::prefix command, 98
!
    command, 146
    operator, 152
!!
    execute previous command, 11
!=
    command, 147
    operator, 152
!N
    execute Nth history event, 11
?
    for non-greedy regular expressions, 210
    in glob patterns, 97, 242
    in regular expressions, 197
?:
    operator, 152
? regexp quantifier, 202
/
    command, 146
    operator, 152
.
    format precision specifier, 86
    in newline-sensitive regular expressions, 209
    in regular expressions, 197, 198
.tclshrc startup file, 10
.wishrc startup file, 13
^
    command, 149
    in bracketed regular expression, 200
    in newline-sensitive regular expressions, 209
    in regular expressions, 197
    operator, 152
    regular expression anchor, 203
^OLD^NEW
    execute previous command, 11
~
    command, 149
    operator, 152
    tilde substitution in paths, 230
(?...) regexp embedded options, 208
(?c) regexp embedded option, 208
(?i) regexp embedded option, 208
(?n) regexp embedded option, 209
(?p) regexp embedded option, 209
(?w) regexp embedded option, 209
(?x) regexp embedded option, 212
(...)
    in regular expressions, 197
()
    for grouping regular expression, 201
    in expressions, 153
[:alnum:] regexp class, 200
[:alpha:] regexp class, 200
[:blank:] regexp class, 200
[:cntrl:] regexp class, 200
[:digit:] regexp class, 200
[:graph:] regexp class, 200
[:lower:] regexp class, 200
[:print:] regexp class, 200
[:punct:] regexp class, 200
[:space:] regexp class, 200
[:upper:] regexp class, 200
[:xdigit:] regexp class, 200
[...]
    in glob patterns, 97, 242
    in newline-sensitive regular expressions, 209
    in regular expressions, 197, 199
[^...]
    in regular expressions, 197
[a-z]
    in regular expressions, 197
{}
    in glob patterns, 242
{*}, 31-31
{M,} regexp quantifier, 202
{M,N} regexp quantifier, 202
{M} regexp quantifier, 202
{n,m}
    in regular expressions, 197
{n}
    in regular expressions, 197
{vbar}
    command, 149
@
    binary format cursor positioning, 166
    binary scan specifier, 171
$
    in newline-sensitive regular expressions, 209
    in regular expressions, 197
    regular expression anchor, 203
${}, 47, 48
$ variable reference, 47
*
    binary format field count, 162
    binary scan field count, 167
    command, 146
    format precision specifier, 86
    in glob patterns, 97, 242
    in regular expressions, 197
    in scan XPG3 specifier, 92
    minimum format field width, 85
    operator, 152
**
    command, 146
    operator, 152
***=
    regular expression literal matching, 208
* regexp quantifier, 202
\
    in glob patterns, 97, 242
    in regular expressions, 197
    shorthands for character classes, 201
    substitution sequences, 25
\a, 25
\A
    regexp constraint escape, 204
\b, 25
\d regexp class, 201
\D regexp class, 201
\f, 25
\m
    regexp constraint escape, 204
\M
    regexp constraint escape, 204
\n, 25
\N regexp back reference, 205
\OOO, 25
\r, 25
\s regexp class, 201
\S regexp class, 201
\t, 25
\u, 25
\U, 25
\v, 25
\w regexp class, 201
\W regexp class, 201
\x, 25
\y
    regexp constraint escape, 204
\Y
    regexp constraint escape, 204
\Z
    regexp constraint escape, 204
&
    command, 149
    operator, 152
&&
    operator, 152
#
    as procedure name, 34
    as variable name, 34
    comments in regular expression, 211
    format flag modifier, 86
    starting a comment, 34
%
    command, 146
    for formatting times, 218
    format conversion specifier, 82
    operator, 152
    scan conversion specifier, 88
%%
    format specifier, 83
    scan specifier, 89
%% time format, 222
%+ time format, 222
%a
    format specifier, 84
%A
    format specifier, 84
%a time format, 219
%A time format, 219
%b
    format specifier, 83
    scan specifier, 89
    time format, 219
%B time format, 219
%c
    format specifier, 83
    scan specifier, 89
    time format, 219
%C time format, 219
%d
    format specifier, 83
    scan specifier, 89
    time format, 219
%D time format, 219
%e
    format specifier, 84
    scan specifier, 89
    time format, 219
%E
    format specifier, 84
    scan specifier, 89
%Ec time format, 219
%EC time format, 220
%EE time format, 220
%Ej time format, 220
%EJ time format, 220
%Es time format, 220
%Ex time format, 219
%EX time format, 219
%Ey time format, 219
%EY time format, 219
%f
    format specifier, 84
    scan specifier, 89
%g
    format specifier, 84
    scan specifier, 89
    time format, 220
%G
    format specifier, 84
    scan specifier, 89
%G time format, 220
%h
    format size modifier, 87
    scan size modifier, 93
    time format, 220
%H time format, 220

%i
format specifier, 83
%I
format size modifier, 87
%I32
format size modifier, 87
%I64
integer size modifier, 87
%I time format, 220
%j
format size modifier, 87
scan size modifier, 93
%j time format, 220
%J time format, 220
%k time format, 220
%l
format size modifier, 87
scan size modifier, 93
time format, 220
%L
format size modifier, 87
scan size modifier, 93
%ll
format size modifier, 87
scan size modifier, 93
%m time format, 220
%M time format, 221
%n
scan specifier, 90
%N time format, 220
%o
format specifier, 83
scan specifier, 89
%Od time format, 221
%Oe time format, 221
%OH time format, 221
%OI time format, 221
%Ok time format, 221
%Ol time format, 221
%Om time format, 221
%OM time format, 221
%OS time format, 221
%Ou time format, 221
%Ow time format, 221
%Oy time format, 221
%p
format specifier, 83
%p time format, 221
%P time format, 221
%q
format size modifier, 87
scan size modifier, 93
%Q time format, 221
%r time format, 221
%R time format, 221
%s
format specifier, 83
scan specifier, 89
time format, 221
%S time format, 221
%t
format size modifier, 87
scan size modifier, 93
%t time format, 221
%T time format, 221
%u
format specifier, 83
scan specifier, 89
time format, 221
%U time format, 221
%V time format, 221
%w time format, 221
%W time format, 221
%x
format specifier, 83
scan specifier, 89
time format, 221
%X
format specifier, 83
scan specifier, 89
time format, 221
%y time format, 222
%Y time format, 222
%z
format size modifier, 87
scan size modifier, 93
%z time format, 222
%Z time format, 222
+
command, 146
format flag, 86
in regular expressions, 197
operator, 152
+ regexp quantifier, 202
<
command, 147
input redirection from files, 466
operator, 152
<@
input redirection from channel, 467
<<
command, 149
input redirection from values, 468
operator, 152
<=
command, 147
operator, 152
<cloned> method
oo::object class, 431
<constructor>, 427
<destructor>, 427
==
command, 147
operator, 152
>
command, 147
operator, 152
writing stdout to a file, 469
>@
writing stdout to a channel, 469
>&
writing stdout and stderr, 470
>&@
writing stdout and stderr, 470
>=
command, 147
operator, 152
>>
appending stdout to a file, 469
command, 149
operator, 152
>>&
appending stdout and stderr, 470
|
in regular expressions, 197, 203
operator, 152
process pipeline, 464
|&
process pipeline, 464
||
operator, 152
0
format flag, 86
0-prefixed integer representation, 142
2>
writing stderr to a file, 469
2>@
writing stderr to a channel, 469
2>@1
returning stderr as command result, 470
2>>
appending stderr to a file, 469
-
command, 146
format flag, 86
operator, 152
-accessPath
Safe Tcl option, 548
-all
lsearch command, 122
regexp command, 206, 207
vwait command, 452
-archive
file attribute command, 238
-ascii
lsearch command, 121
lsort command, 115
-async
socket command, 522
-autoPath
Safe Tcl option, 548
-base
clock scan command, 224
-bisect
lsearch command, 123
-blocking
chan configure command, 488
-buffering
chan configure command, 255
-buffersize
chan configure command, 256
-calibrate
timerate command, 613
-checksum
chan configure command, 506
-clear
namespace export command, 352
-code
return command, 320
-command
fcopy command, 266
interp limit command, 551
namespace which command, 357
regsub command, 213
-connecting
chan configure command, 526
-creator
file attribute command, 238
-decreasing
lsearch command, 123
lsort command, 116
-deleteHook
Safe Tcl option, 548
-dictionary
chan configure command, 506
lsearch command, 121
lsort command, 115
-direct
timerate command, 614
-directory
glob command, 244
-during, 329
-encoding
chan configure command, 262, 264
source command, 68
-eofchar
chan configure command, 261
-error
chan configure command, 526
tcl::prefix match command, 99
-errorcode
return command, 333
return options dictionary, 328, 332
-errorinfo
return command, 333
return options dictionary, 327, 332
-errorline
return options dictionary, 327
-errorstack
return command, 333
return options dictionary, 328
-exact
lsearch command, 120
switch command, 62
tcl::prefix match command, 99
-exactlocale
mcexists command, 192
-exactnamespace

mcexists command, 192
-expanded
regexp command, 211
regsub command, 213
-extended
vwait command, 453
-failindex
encoding convertfrom command, 184
encoding convertto command, 183
string is command, 94
-flush
chan configure command, 506
-force
file delete command, 240
namespace import command, 352
-format
clock format command, 218
clock scan command, 222
-get
property command, 437
-glob
lsearch command, 120
switch command, 62
-global
load command, 383
-gmt
clock format command, 218
-granularity
interp limit command, 551
-group
file attribute command, 237
-handshake
chan configure command, 529
-hard
file link command, 247
-header
chan configure command, 506
-hidden
file attribute command, 238, 238
-ignorestderr
exec command, 472
-increasing
lsearch command, 123
lsort command, 116
-index
lsearch command, 121
-indexvar
switch command, 62
-indices
regexp command, 207
-inline
lsearch command, 123
regexp command, 207
-inputmode
chan configure command, 260
-integer
lsearch command, 121
lsort command, 115
-join
glob command, 245
-keepalive
chan configure command, 526
-keepnewline
exec command, 463
-kind
property command, 437
-lasterror
chan configure command, 530
-lazy
load command, 383
-length
string compare command, 94
string equal command, 93
-level
return command, 320, 570
-limit
chan configure command, 506
-line
regexp command, 209
regsub command, 213
-lineanchor
regexp command, 209
regsub command, 213
-linestop
regexp command, 209
regsub command, 213
-locale
clock add command, 225
clock format command, 218
clock scan command, 223
-longname
file attribute command, 238
-map
namespace ensemble configure command, 362
-matchvar
switch command, 62
-message
tcl::prefix match command, 98
-mode
chan configure command, 529
-myaddr
socket command, 521
-myport
socket command, 521
-namespace
mcexists command, 192
-nested
Safe Tcl option, 548
-nestedLoadOk
Safe Tcl option, 548
-nobackslashes
subst command, 82
-nocase
lsearch command, 120
lsort command, 115
regexp command, 208
regsub command, 213
string compare command, 94
string equal command, 93
string map command, 77
string match command, 97
-nocommands
subst command, 82
-nocomplain
glob command, 243
-nodelay
chan configure command, 526
-nofilevents
vwait command, 452
-noidleevents
vwait command, 452
-nonewline
chan puts command, 255
-noStatics
Safe Tcl options, 548
-not
lsearch command, 120
-notimerevents
vwait command, 452
-novariables
subst command, 82
-nowindowevents
vwait command, 452
-options
return command, 320
-overhead
timerate command, 614
-owner
file attribute command, 237
-parameters
namespace ensemble configure command, 363
-path
glob command, 244
-peername
chan configure command, 526
-permissions
file attribute command, 237
-pollinterval
chan configure command, 530
-prefixes
namespace ensemble configure command, 363
-profile
chan configure command, 262
encoding convertfrom command, 185
encoding convertto command, 183
-queue
chan configure command, 529
-readable
vwait command, 452
-readonly
file attribute command, 237, 238, 238
-real
lsearch command, 121
lsort command, 115
-regexp
lsearch command, 120
switch command, 62
-rsrclength
file attribute command, 238
-safe
interp create command, 540
-set
property command, 437
-shortname
file attribute command, 238
-size
fcopy command, 266
-sockname
chan configure command, 526
-sorted
lsearch command, 123
-start
lsearch command, 124
regexp command, 210
regsub command, 213
-statics
Safe Tcl options, 548
-stride
lsearch command, 122
-subcommands
namespace ensemble configure command, 361
-subindices
lsearch command, 122, 123
-symbolic
file link command, 247
-sysbuffer
chan configure command, 529
-system
file attribute command, 238
-tails
glob command, 245
-timeout
chan configure command, 530
vwait command, 452, 453
-timezone
clock add command, 225
clock format command, 218
clock scan command, 223
-translation
chan configure command, 261, 264
-ttycontrol
chan configure command, 530
-ttystatus
chan configure command, 530
-type
glob command, 243
-unknown
namespace ensemble configure command, 364
-validate
clock scan command, 224
-value
interp limit command, 551

-variable
namespace which command, 357, 358
vwait command, 452
-winsize
chan configure command, 260
-writable
vwait command, 452
-xchar
chan configure command, 530

A

a
binary format specifier, 163
binary scan specifier, 169
A
binary format specifier, 163
binary scan specifier, 169
abs command, 156
abstract classes, 434-435
access modes
string form, 250
access modes, list form, 251
acos command, 156
actors, 588
Adler-32, 179
after command, 456-461
cancel, 460-460
idle queue, 457-460
info, 461-461
scheduling, 456-457
sleeping, 456-456
aliases, 534-537
defining, 534-535
deleting, 536
introspection, 536
anchors
in regular expressions, 203-203
append command, 76, 76
apply command, 43-45
argc global variable, 17
argument expansions, 31-31
arguments, 40
(see also parameters)
defaults, 40
expanding, 31-31
formal, 40
interpretation, 32
introspection, 45
variable number, 41-41
argv0 global variable, 17
argv global variable, 17
arithmetic
operations, 145-155
array
printing, 58
array command
anymore subcommand, 57
default subcommand, 53, 53
donesearch subcommand, 57
exists subcommand, 54
for subcommand, 56
get subcommand, 55
nextelement subcommand, 57
set subcommand, 54
size subcommand, 57
startsearch subcommand, 57
statistics subcommand, 57
array names command, 54
arrays, 52-59
associative, 52
checking element existence, 54
checking existence, 54
compared to dictionaries, 138-139
element count, 57
empty strings as keys, 58
enumerating elements, 54
internal statistics, 57
iterating with foreach, 56
key comparisons, 58
keys, 53
multi-dimensional, 58
multiple element assignment, 54
operating on multiple elements, 54
referencing elements, 52
retrieving multiple elements, 55
size, 57
tracing, 291-293
unsetting elements, 55
whitespace in keys, 58
asin command, 156
asynchronous I/O (see I/O, asynchronous)
atan2 command, 156
atan command, 156
atoms
in regular expressions, 201-202
auto_execok command, 33, 466
auto_load command, 33, 374
search path, 378
auto_mkindex command, 375
auto_noexec global variable, 11
auto_noload global variable, 33
auto_path global variable, 378
in Safe Tcl, 549
autoloading, 374

B

b
binary format specifier, 164
binary scan specifier, 169
B
binary format specifier, 164
binary scan specifier, 169
background errors, 461-462
background tasks
exception processing, 461-462
back references
in regular expressions, 205-206
backslash (see \)
escape sequences, 25
newline within braces, 29
sequences, 25-26
base-2 representation, 142
base64 encoding
of binary strings, 160
bgerror command, 462
Big5, 181
bignum integers
converting to, 144
bin2hex, 159
bin2hex command, 618
binary command
decode subcommand
base64 subcommand, 160
hex subcommand, 160
uuencode subcommand, 161
encode subcommand
base64 subcommand, 160
hex subcommand, 160
uuencode subcommand, 161
format subcommand, 161-166
count field, 162
cursor positioning, 166
type specifiers, 163-166
u flag, 162
scan subcommand, 166-171
count field, 167
cursor positioning, 171
type specifiers, 168-171
u flag, 167
binary commmand
decode hex subcommand
-strict option, 160
decode subcommand
base64 -strict option, 161
encode subcommand
base64 -maxlen option, 161
base64 -wrapchar option, 161
binary distributions, 6-6
binary strings
base64 encoding, 160
constructing, 161-166
hexadecimal encoding, 160
literals, 159
parsing, 166-171
uuencode encoding, 161
blocking I/O (see I/O, blocking)
bool command, 144, 156
booleans
converting to, 144
braces
for quoting strings, 29
nesting, 30
brackets, 27
break command, 64
emulating with return, 324
buffering (see channels, buffering)
build-info command
in namespace tcl, 20
building
on Linux/Unix, 7
on Windows, 8

C

c
binary format specifier, 165
binary scan specifier, 170
C
binary scan specifier, 169
calendars
Gregorian, 216
Julian, 215
locale-specific, 216
callback command, 401
callbacks
command tracing, 296
execution tracing, 297
scripts vs. procedures, 45
specifying namespace contexts, 346-347
to methods, 401-401
variable tracing, 288
call by name, 275
call frames, 269-288
commands creating, 272
executing scripts in, 276-280
inspecting with info level, 270-271
introduction, 269
linking variables, 272-275
retrieving command for, 271
call stack, 269-270
determining depth, 270
in event handlers, 455-456
specifying levels, 272
unwinding, 321
capturing parenthesis
in regular expressions, 201
case sensitivity
in regular expressions, 208
in string compare, 94
in string equal, 93
in string map, 77
catch command, 326-326, 326
cd command, 18
ceil command, 156
chan command, 249
blocked subcommand, 488, 489
(see fblocked)
close subcommand, 254
configure subcommand, 506
sockets, 526-527
copy subcommand (see fcopy)
create subcommand, 515

eof subcommand, 258, 489
event subcommand (see fileevent command)
flush subcommand (see flush)
gets subcommand
blocking mode, 257-257
non-blocking mode, 488-491
isbinary subcommand, 264
names subcommand, 267
pending subcommand, 489, 490
pipe subcommand, 479-482
pop subcommand, 504-505
postevent subcommand, 511
push subcommand, 504-505
puts subcommand, 255
read subcommand
blocking mode, 257-258
non-blocking mode, 491-492
seek subcommand, 265-266
tell subcommand, 264-265
truncate subcommand, 266
chan configure command, 476
channel
events (see events, channel)
channels, 249
access pointer, 264-266
binary I/O, 263-264
buffering, 254, 255
buffer size, 256
closing, 254
configuration, 254, 526
copying data, 266
creating
files, 250-253
encoding, 262-262
encoding profiles, 262-263
end-of-line characters, 260
enumerating, 267
half-closing, 254, 483-483
in child interpreters, 539-540
input buffering, 259
reading, 256-259
blocking mode, 257, 257
characters, 257-258
lines, 257-257
reflected, 507-520
socket options, 526
standard I/O, 249-250
on Windows, 250
transforms (see channel transforms)
writing to, 255-256
channel tranforms
buffering, 503
channel transforms, 498-507
clear subcommand, 503
drain subcommand, 503
finalize subcommand, 501
flush subcommand, 503
implementing, 499-504
initialize subcommand, 501
limit? subcommand, 504
read subcommand, 502
using, 504-505
write subcommand, 502
zlib, 505-507
character case
changing, 79
in list searches, 120
character classes
in regular expressions, 199, 199
character encodings, 181-186, 181
(see also encoding command)
adding, 185
converting from, 184
converting to, 183
encoding profiles (see profiles)
enumerating, 183
system encoding, 185
characters
indexing in strings, 75
checksum method
zlib streams, 177
checksums
Adler-32, 179
CRC-32, 179
zlib streams, 177
classes
abstract, 434-435
as filter mix-ins, 423
base, 406
creating, 392-392
definition script, 393-394
derived, 406
destroying, 394-394
enumerating, 438
inheritance (see inheritance)
instances (see objects)
introspection
relationships, 439
mix-ins (see mix-ins)
modifying, 403
namespaces, 396
oo::abstract, 434
oo::class, 438
oo::configurable, 436
oo::object, 439
oo::singleton, 435
class variable, 394, 395-396
cleanupTests command
in namespace tcltest, 609
C locale, 187
clock::rfc2822 package, 227
clock command
add subcommand, 225
clicks subcommand, 217
format subcommand, 217-222
microseconds subcommand, 217
milliseconds subcommand, 217
scan subcommand, 222
seconds subcommand, 217
close (see chan command, close)
close command
half-closing, 483
close method
zlib streams, 178
code points
Unicode, 73-74
command ensembles (see ensemble commands)
command history, 309-312
interactive mode, 11
command history events, 310
command line
editing
rlwrap, 12
tclreadline, 12, 12
command-line arguments, 17-18
commands
abbreviating, 33
aliases, 534-537 (see aliases)
bracketed, 27
checking existence, 38
constructing names, 279
deleting traces, 299
enumerating, 37-38
handling unknown commands, 33-34
hidden, 542-545, 545
introduction, 32-46
invocation, 32-34
invocation count, 312
invoking through variables, 32
one-time initialization, 36
raising exceptions, 332-334
redefining, 36
renaming, 35-36
substitution, 27
tracing, 296-299
tracing execution, 297-299
tracing renaming and deletion, 296-297
type, 38-39
types, 38
wrapping, 35
wrapping using tailcall, 283
command separators, 23
comments
in expressions, 150-151
syntax, 34-35
communicating sequential processes, 588
compression (see data compression)
concat command, 113
conditional evaluation
if command, 60
conditional evaluration
switch command, 61
configure method
oo::configurable class, 437
console configuration, 260-260
const command, 59
constraint escapes
in regular expressions, 204
constraints
in regular expressions, 203-205
lookahead in regular expressions, 204
constructor method, 396
contexts
execution, 269-288
private, 410-414
variable, 269-288
continue command, 65
control statements
custom, 278, 336
conversions
numeric, 144-145
conversion specifier
in format command, 82
in scan command, 88
cookiejar package, 615
copy command
in namespace oo, 431
coroinject command, 576
coroprobe command, 576
coroutine::util command
after subcommand, 588
gets subcommand, 588
coroutine command, 564
coroutine package, 587
coroutines
as generators, 579-582
creating, 564
private variables, 574-575
suspending, 564-569
cos command, 156
cosh command, 156
country codes, 186
CRC-32, 179
create command
in namespace pkg, 381
in pkg namespace, 381
create method, 403
CSP (see communicating sequential processes)
C stack, 280-282
preserving, 282-286
current
locale, 218

D

d
binary format specifier, 166
binary scan specifier, 171
databases (see TDBC)
data compression, 172-179
data flow programming

using traces, 294-295
data members
in derived classes, 408
dde package, 615
deadlock
with vwait, 453
decimal representation, 142
define command
in namespace oo, 392
definitionnamespace
retrieving, 440-440
definitionnamespace command, 433
DEFLATE algorithm, 173, 174
deletemethod command, 400, 416
destroy method, 394, 404
destructor method, 396
dict command
append subcommand, 131
create subcommand, 126
exists subcommand, 127
filter subcommand, 134
for subcommand, 133
getdef subcommand, 128
get subcommand, 128
getwithdefault subcommand, 128
incr subcommand, 132
info subcommand, 138
keys subcommand, 129
lappend subcommand, 131
map subcommand, 133
merge subcommand, 132
remove subcommand, 131
replace subcommand, 132
set subcommand, 129
size subcommand, 138
unset subcommand, 130
update subcommand, 135
compared to with, 137
values subcommand, 129
with subcommand, 136
compared to update, 137
dictionaries
appending list elements, 131
appending to elements, 131
checking key existence, 127
compared to arrays, 138-139
enumerating keys, 129
enumerating values, 129
filtering elements, 134-135
from lists, 127
incrementing elements, 132
iterating, 133-133
looking up keys, 128
mapping elements, 133
nested, 126
removing elements, 130
replacing values, 132
setting a nested value, 130
setting a value, 129-130
shadowing nested dictionaries, 136-138
shadowing with local variables, 135-136
size, 138
statistics, 138
string representation, 125
dir
global variable, 381
directories, 239
(see also files)
creating, 239
working directory, 18-18
dodekalogue, 23
domain specific language (see DSL)
double command, 144, 156
double quotes
for quoting strings, 28
double quotes, escaping, 29
double substitution, 66
protecting against, 67
double substitutions
in eval, 66-67
in uplevel, 276
Drucker
Peter, 215
DSL implementation, 555-561

E

EIAS, 71
EIAS principle, 71-71
encoding command, 181
(see also character encodings)
convertfrom subcommand, 184
convertto subcommand, 183
dirs subcommand, 185
names subcommand, 183
profiles subcommand, 183
encoding profiles, 182-183
in channel I/O, 262-263
replace, 182
strict, 182
tcl8, 182
encodings (see character encodings)
end
list index, 104
end,
in linsert command, 107
ensemble commands, 39-39, 359-371
as objects, 369-371
checking for, 366
configuring, 361-364
configuring subcommands, 361-362
creating, 359-361
defined, 359
enabling prefixes, 362-363
for defining accessors, 368-369
modifying existing, 367-368
nesting, 366-371
unknown commands, 364-366
entier command, 144, 156
env global variable, 18, 483
environment
passing to child processes, 483
environment variables
HOME, 230
LANG, 187
LC_ALL, 187
LC_MESSAGES, 187
PATH, 465
TCL_LIBRARY, 373, 373
TCL_MAJOR_MINOR_TM_PATH, 386
TCL_PKG_PREFER_LATEST, 380
TCL_TZ, 216
TCLLIBPATH, 378
TZ, 216
EOF
detecting, 257, 258-259
eof (see chan command, eof)
EOF character, 261
eof command, 491
epoch
date, 215
eq
command, 148
operator, 152
errorCode global variable, 328, 332
error command, 332-333
error handling
in process pipelines, 479
errorInfo global variable, 70, 327, 332
errors
details, 327-328
in exec command, 471-473
message, 327-328
recovering from, 329-332
traceback, 327-328
eval command, 65-67
event driven I/O
in reflected channels, 510-512
event-driven I/O, 493-496
event handlers
call stack, 455
event loop, 450-456
exception processing, 461-462
introduction, 450
invoking once, 454-455
operation, 450
starvation, 458-460
using vwait, 451-453
event queue, 450
events
channel, 449, 493-496
idle, 449, 457-460
readable, 493
sources, 449
timer, 449, 456-457
writeable, 493
exceptions
error exceptions (see error)
forwarding
with error, 335-335
handling with try, 329-332
raising, 332-334
trapping with catch, 326
exec command, 463-474
creating background processes, 474
encoding, 474
error handling, 471-473
limitations, 474-474
using with pipes, 481
exit code, 15
from child process, 472
exit command, 10, 15
in Safe Tcl, 547
exp command, 156
export command, 398
expr command, 149-154, 613
comments, 150-151
operand syntax, 151-151
operators, 151-153
expressions, 149-154
grouping, 153
operands, 151-151
operators, 151-153
extensions, 382-384
external programs
automatically running, 11

F

f
binary format specifier, 165
binary scan specifier, 170
fblocked command, 488
fconfigure command (see chan command, configure)
fcopy command, 266
field specifiers
in format command, 82
file
metadata, 234
file command, 229
atime subcommand, 234
attribute subcommand, 237-239
copy subcommand, 240
delete subcommand, 240
dirname subcommand, 231
executable subcommand, 236
exists subcommand, 236
extension subcommand, 231
home subcommand, 230
isdirectory subcommand, 237
isfile subcommand, 237
join subcommand, 232

link subcommand, 247
lstat subcommand, 234
mkdir subcommand, 239
mtime subcommand, 234
nativename subcommand, 233
normalize subcommand, 232
owned subcommand, 236
pathtype subcommand, 230
readable subcommand, 236
readlink subcommand, 248
rename subcommand, 240
rootname subcommand, 231
separator subcommand, 239
size subcommand, 234
split subcommand, 231
stat subcommand, 234
system subcommand, 239
tail subcommand, 231
tempdir subcommand, 248
tempfile subcommand, 248
tildeexpand subcommand, 230
type subcommand, 236
volumes subcommand, 239
writable subcommand, 236
fileevent command, 493-496
file paths
absolute, 230
constructing, 232
extension, 231
home directory, 230, 230
native format, 233
normalization, 232-233
parsing, 231-231
relative, 230
syntax, 229-229
volume relative, 230
files
access time, 234, 235
attributes, 237-239
on macOS, 238
on Unix, 237
on Windows, 237
case sensitivity, 246
check if directory, 237
check if regular file, 237
checking access permission, 235-236
checking existence, 236
checking ownership, 236
copying, 240-242
creating, 252
creation time, 235
deleting, 240
device id, 235
enumerating, 242-247, 246
based on patterns, 242
by attributes, 243
by type, 243
hidden files, 246
recursing into directories, 246
executing scripts, 68-70
group id, 235
hard links, 235, 247
inode number, 235
links, 247-248
metainformation, 234-235
modification time, 234, 235
opening, 250-253
renaming, 240-242
size, 234, 235
soft links, 247
tclIndex, 374
temporary, 248-248
timestamps, 234-234
truncating, 266
type, 235
types, 236-237
user id, 235
file system
path separator, 239
retrieving volumnes, 239
file systems
type, 239
fileutil::find command, 581
fileutil package, 246
filter command, 422
filter methods, 422-424
enumeration, 445
introspecting, 445-446
using mix-ins, 423
filter programs
using open, 481
finalize method
zlib streams, 176
finally
try command, 329, 331
find command
in namespace fileutil, 581
floating point
converting to, 144
precision in formatting, 86
floating point types, 142
floor command, 156
flush command, 256
in non-blocking mode, 492
flush method
zlib streams, 179
fmod command, 156
for command, 64
foreach command, 106
iterating over arrays, 56
foreachLine command, 260
formal arguments, 40
format command, 82-87
conversion characters, 83
field specifiers, 82
integer specifiers, 83
minimum field widths, 85
precision specifier, 86
size modifier, 87
specifier flags, 85
string specifiers, 83
XPG3 position specifiers, 84
formatting
binary strings, 161-166
text, 82-87
forward command, 420
forwards
private, 412-413
fpclassify command, 143
frames (see call frames)
fullflush method
zlib streams, 179
fullnormalize command, 233
functions
custom, 157
in expressions, 156

G

ge
command, 148
operator, 152
generator package, 582
generators
based on coroutines, 579-582
generic command
in namespace platform, 389
in namespace platform::shell, 389
get method
zlib streams, 176, 178
gets (see chan command, gets)
geturl command
in namespace http, 527
global command, 50
globals
tcl_interactive, 478
tcl_precision, 144
global variable
auto_path, 378
global variables, 50-50
argc, 17
argv, 17
argv0, 17
auto_noexec, 11
auto_noload, 33
auto_path, 549
dir, 381
env, 18, 483, 540
errorCode, 328, 332
errorInfo, 70, 327, 332
in Safe Tcl, 549
tcl_interactive, 12
tcl_library, 373, 378, 603
tcl_nonwordchars, 80
tcl_patchlevel, 20
tcl_pkgPath, 378
tcl_platform, 21
tcl_version, 19
tcl_wordchars, 80
glob command, 242-247, 243
case sensitivity, 246
patterns, 242
short names on Windows, 246
glob patterns, 96, 242
for unsetting array elements, 55
in enumerating procedures, 45
in enumerating variables, 51
in string match, 96
selecting array elements, 54, 55, 58
Gregorian calendar, 216
groups
in regular expressions, 201-202, 201
gt
command, 148
operator, 152
gzip compression, 172
GZIP compression, 174-174

H

h
binary format specifier, 164
binary scan specifier, 169
H
binary format specifier, 164
binary scan specifier, 169
handlers
try command, 329
header method
zlib streams, 178
hexadecimal representation, 142
of binary strings, 160
hidden commands (see commands, hidden)
history
recall patterns, 33
history command, 11, 309-312
add subcommand, 311
change subcommand, 311
clear subcommand, 311
event subcommand, 310
info subcommand, 310
keep subcommand, 311
nextid subcommand, 310
redo subcommand, 311
HOME, 230
home directory, 230
HTML parser, 305
HTTP, 615
http::geturl command, 527
http::register command, 528
http package, 527, 615
HTTP protocol, 527
HTTPS protocol, 528
hypot command, 156

I

i
  binary format specifier, 165
  binary scan specifier, 170
I
  binary format specifier, 165
  binary scan specifier, 170
I/O
  asynchronous, 487-497
  binary, 263-264
  blocking, 487
  non-blocking, 487, 488-493
    read operations, 488-492
  non-blocking mode
    reading characters, 491-492
    reading lines, 488-491
    writing, 492
  redirection, 466-470
identifiers
  automatic generation, 312
  variable, 46-47
identify (see platform::shell::identify)
identify command
  in namespace platform, 388
  in namespace platform::shell, 389
identity function, 62, 321
idle task queue, 450
idle tasks
  adding, 457-460
if command, 60
in
  command, 148
  operator, 152
incr command, 155-155, 155
indices
  for nested lists, 104
  list, 104-104
  strings, 74
Inf, 142
infinity, 142
info command, 70
  args subcommand, 45
  body subcommand, 46
  class subcommand, 438
    call subcommand, 426, 445
    constructor subcommand, 444
    definitionnamespace subcommand, 440
    definition subcommand, 444
    destructor subcommand, 444
    filters subcommand, 445
    forward subcommand, 444
    instances subcommand, 438, 438
    methods subcommand, 443
    methodtype subcommand, 443
    mixins subcommand, 440
    properties subcommand, 447
    subclasses subcommand, 440
    superclasses subcommand, 439
    variables subcommand, 447
  cmdcount subcommand, 312, 551
  cmdtype subcommand, 38
  complete subcommand, 24
  constant subcommand, 59
  consts subcommand, 59
  coroutine subcommand, 569, 586
  default subcommand, 45
  errorstack subcommand, 328
  executable subcommand, 386
  exists subcommand, 52
  frame subcommand, 286
  functions subcommand, 156
  globals subcommand, 51
  level subcommand, 270-271
  library subcommand, 373
  loaded subcommand, 384
  locals subcommand, 51
  nameofexecutable subcommand, 19
  object subcommand, 438
    call subcommand, 425, 426
    class subcommand, 441
    creationid subcommand, 441
    definition subcommand, 444
    filters subcommand, 445
    forward subcommand, 444
    isa class subcommand, 439
    isa metaclass subcommand, 439
    isa mixin subcommand, 442
    isa object subcommand, 442
    isa typeof subcommand, 442
    methods subcommand, 443
    methodtype subcommand, 443
    mixins subcommand, 442
    namespace subcommand, 441
    properties subcommand, 447
    variables subcommand, 447
    vars subcommand, 447
  patchlevel subcommand, 20
  procs subcommand, 45
  script subcommand, 68
  sharedlibextension subcommand, 383
  tclversion subcommand, 19
  vars subcommand, 51
info commands, 37-38
info exists command, 54
inheritance, 406-414
  multiple, 409-410
  vs. mix-ins, 420
init command
  in namespace profiler, 611
initialise command, 403
initialization
  lazy, 36, 293
initialize command, 395, 403
input-output (see I/O)
input validation, 94-96
installing Tcl
  binary distributions, 6-6
instance variable, 394, 394-395
int command, 144, 156
integers
  base-2 representation, 142
  converting to, 144
  decimal representation, 142
  hexadecimal representation, 142
  incrementing, 155-155
  octal representation, 142
interactive mode
  command history, 11
  detecting, 12
  tclsh shell, 10-12
  wish shell, 13-14
interpAddToAccessPath command
  in namespace safe, 549
interp command
  aliases subcommand, 536
  alias subcommand, 534, 536
    for safe interpreters, 541-542
  bgerror subcommand, 462-462
  cancel subcommand, 537
  children subcommand, 533
  create subcommand, 531
  debug subcommand, 288
  delete subcommand, 533
  eval subcommand, 533
  exists subcommand, 533
  expose subcommand, 544
  hidden subcommand, 545
  hide subcommand, 544
  invokehidden subcommand, 542
  limit subcommand, 552
    command subcommand, 551
  marktrusted subcommand, 545
  recursionlimit subcommand, 550
  share subcommand, 539
  target subcommand, 536
  transfer subcommand, 539
interpConfigure command
  in namespace safe, 548
interpCreate command
  in namespace safe, 547, 549
interpDelete command
  in namespace safe, 547
interpFindInAccessPath command
  in namespace safe, 549
interpInit command
  in namespace safe, 547, 549
interpolation, 29
interpreter
  identification, 532-532
interpreters
  child, 531
  creating, 531
  destroying, 533
  enumerating children, 533
  parent, 531
  paths, 532-532
  safe (see safe interpreters)
introspection, 70
  aliases, 536
  arguments, 45
  classes, 438-448
  filter methods, 445-446
  name resolution, 357-359
  namespaces hierarchies, 343-343
  objects, 438-448
  parameter defaults, 45
  parameters, 45
  procedure bodies, 46
  procedures, 45-46
  properties, 447-448
  traces, 300
  variables, 447-447, 447
isfinite command, 156
isinf command, 156
island package, 549
isnan command, 156
isnormal command, 156
ISO 3166 standard, 186
ISO 639 standard, 186
ISO 8601, 227 (see time, ISO 8601 format)
iso8601 package, 227
ISO8859-1, 181
isqrt command, 156
issubnormal command, 156
isunordered command, 156
Itcl package, 616
iteration, 64
  (see also looping)

J

join command, 79
Julian calendar, 215

L

lambda command, 44, 618
lambda package, 45
LANG, 187
language codes, 186
lappend command, 107
lassign command, 105
LC_ALL, 187
LC_MESSAGES, 187
le
  command, 148
  operator, 152
leap seconds, 215
ledit command, 109
levels, 270, 270
  (see also call frames)

retrieving, 270-271
specifying for info level, 271
specifying for uplevel, 276
specifying for upvar, 272
lexicographical ordering, 94
lexnormalize command, 233
libraries
autoloading, 374
search path, 378
lindex command, 104-104
link command, 402, 436
linsert command, 107
list command, 101
lists
appending elements, 107
as dictionaries, 127
concatenating, 113
constructing with regexp, 110
constructing with split, 110-111
count of elements, 110
defined, 101
extracting leading elements, 105
from repeated elements, 113
indexing elements, 104
indices, 104-104
inserting elements, 107
literals, 101-103
looping, 106-107
nested, 104, 104
numeric sequences, 111
parallel
looping, 106
removing duplicates, 119-119
replacing elements, 109
retrieving sublists, 105
reversing, 114
search from an offset, 124
searching, 120-124
searching groups, 122-122
searching nested lists, 121-122
searching sorted lists, 123
setting elements, 108
sorted indices, 118
sorting, 115-119, 115
(see also lsort command)
string representation, 101-103
transforming elements, 113-114
literals
dictionaries, 125
lists, 101-103
llength command, 110
lmap command, 113-114
load command, 383
locales, 186
(see also localization)
getting system locale, 190
retrieving and setting, 190
localization, 186
(see also locales)
directories, 187
using XPG3 position specifiers, 85
log10 command, 156
log command, 156
looping
terminating with break, 64
lpop command, 105
lrange command, 105
lremove command, 108
lrepeat command, 113
lreplace command, 109
lreverse command, 114
lsearch command, 120-124
matching options, 120-121
lseq command, 111-112
lset command, 108
lsort command, 115-119
-command option, 44, 115
-index option, 117-117
-indices option, 118-119
-stride option, 117-118
-unique option, 119-119
lt
command, 148
operator, 152

M

m
binary format specifier, 165
binary scan specifier, 170
Machiavelli, 32
macOS, 8
main script
checking for, 69
maps, 52
mathematical functions, 156-157
mathfunc namespace, 156-157
custom functions, 157
max command, 156
mc, 188
mc command
namespace context, 191
mcexists command, 192
mcflmset command, 188
mcflset command, 188
mcforgetpackage command, 194
mcload command
in namespace msgcat, 188
mcloadedlocales command, 188
mclocale, 190
mclocale command, 190
mcmax command, 189
mcmset command, 187
mcn command, 192
mcpackageconfig command, 194
mcpackagelocale command, 193
mcpackagenamespaceget command, 194
mcpreferences command, 190
mcset command, 187
mcunknown command
in namespace msgcat, 192
mcutil command
getpreferences subcommand, 191
getsystemlocale subcommand, 190
message catalogs, 187-188
creating, 187
directories, 187
loading, 188
missing entries, 192
partitioning with namespaces, 191-192
metaclasses, 431-438
custom, 433-434
oo::abstract, 434-435
oo::configurable, 436-438
oo::singleton, 435-436
metadata
files, 234
metaprogramming, 302-309
method chains, 424-430
method command, 397
methods, 396-402
as callbacks, 401-401
chaining, 408, 424-430
checking end of chain, 427
introspection, 426
introspection in method contexts, 427
search order, 425
unknown methods, 426
chains, 420
context, 405
defining, 397-397
deleting, 400
exported, 397
filters (see filter methods)
forwarding, 420-422
in derived classes, 407
inheriting, 407
introspection
chains, 445
definition, 444
enumeration, 443
invoking, 404
mix-ins, 420
overriding, 407
private, 397, 412-413
renaming, 400
unexported, 397
min command, 156
min-ins
multiple, 419
mixin command, 419
mix-ins, 418-420
for classes, 419
for objects, 419
vs. inheritance, 420
modules, 384-388
creating, 387
file name syntax, 385
installing, 387
removing directories, 387
search path, 385, 387
adding directories, 386
initialization, 386
msgcat::mcload command, 188
msgcat::mcpackageconfig command
get subcommand, 194
isset subcommand, 194
set subcommand, 194
unset subcommand, 194
msgcat::mcpackagelocale command
clear subcommand, 193
get subcommand, 193
isset subcommand, 193
loaded subcommand, 193
preferences subcommand, 193
present subcommand, 193
set subcommand, 193
unset subcommand, 193
msgcat::mcunknown command, 192
msgcat package, 188
multiple inheritance (see inheritance, multiple)
myclass command, 399
my command
for invoking methods, 405
variable subcommand, 395, 408, 416
varname subcommand, 405
mymethod command, 401

N

n
binary format specifier, 165
binary scan specifier, 170
name resolution, 349-359
command names, 351-357
introspecting, 357-359
namespace names, 351
variables, 349-351
outside procedures, 349
within procedures, 350-350
namespace command
children subcommand, 343
code subcommand, 346
current subcommand, 343
delete subcommand, 344
ensemble configure subcommand, 361
ensemble create subcommand, 360
-command option, 360
ensemble exists subcommand, 366
eval subcommand, 345
exists subcommand, 344

export subcommand, 352
forget subcommand, 353
import subcommand, 352, 355
inscope subcommand, 345
origin subcommand, 357
parent subcommand, 343
path subcommand, 354, 355
qualifiers subcommand, 344
tail subcommand, 344
unknown subcommand, 356
upvar subcommand, 351
which subcommand, 357
namespace ensembles (see ensemble commands)
namespaces
absolute, 341
checking existence, 344
context, 340, 340
contexts, 349
creating, 339-345
defining commands in, 348-349
defining variables in, 347-348
definition, 339
deleting, 344
executing code within, 345-347
exporting names, 352-354
for classes, 396
for objects, 405-405
getting caller's context, 279
hierarchies, 341-344
introspection, 343-343
importing names, 352-354
importing versus paths, 355-356
in callbacks, 346-347
in message catalogs, 191
names of, 341-344
nested, 341, 342
relative, 341
scope, 339, 340
search path, 354-355
variables
linking local variables, 351
within procedures, 349
NaN, 143, 148
ne
command, 148
operator, 152
newlines
matching in regular expressions, 209-209
new method, 403
next command, 407
search order, 424
nextto command, 408, 429
ni
command, 148
operator, 152
non-blocking I/O (see I/O, non-blocking)
in reflected channels, 510-512
non-capturing parenthesis
in regular expressions, 202
now
adding time, 225
formatting time, 217
numbers
converting, 144-145
representation, 141-145
numeric sequences, 111
nx, 391

O

objdefine command
in namespace oo, 414, 417
object-oriented programming
classless, 430-431
objects
changing class, 417
creating, 403
definition script, 414-415
destroying, 404
enumerating, 438
identity, 440-441
namespace, 440
namespaces, 405-405
renaming, 441
specialization, 414-418
octal representation, 142
on
try clause, 329
oo::abstract class, 434
oo::class class, 438
oo::class command
create subcommand, 392
oo::configurable class, 436
oo::configuresupport, 438
oo::copy command, 431
oo::define command, 392
oo::objdefine command, 414, 417
oo::object class, 439
oo::singleton class, 435
open command, 250, 481
for filters, 481
for pipelines, 474-479
operator precedence, 152-153
options, 41-42

P

package command
files subcommand, 377
ifneeded subcommand, 381
names subcommand, 377, 384
prefer subcommand, 380
present subcommand, 380
provide subcommand, 382
require subcommand, 379, 384
search path, 378
versions subcommand, 377
packages
cookiejar, 615
coroutine, 587
creation, 381-382
dde, 615
defined, 375
generator, 582
http, 527, 615
island, 549
Itcl, 616
lambda, 45
latest version, 380
naming, 375
platform::shell, 389
profiler, 611
registration, 380-381
registry, 615
stable version, 380
tcl::idna, 195
Thread, 615
Tk, 615
using, 378-380
versions
comparing, 376-377
selecting, 379
syntax, 375-376
vfs, 615
package vcompare command, 376
package vsatisfies command, 376
parameter defaults
introspection, 45
parameters, 40
(see also arguments)
args, 41-41
default values, 40
defining, 40
introspection, 45
named, 41-42
vs. arguments, 40
parenthesis, 153
parray command, 58
parsing
binary strings, 166-171
with scan, 87-93
patch level
obtaining, 20
path
normalization, 69
PATH, 465
pattern matching
for searching lists, 120
glob, 96
patterns, 197
(see also regular expressions)
patterns command
in namespace platform, 389
PID, 19 (see processes, identifiers)
pid command, 19
pipeline
pipeline (see processes)
pipes, 479-482
pkg_mkIndex command, 381
pkg::create command, 381
pkgconfig command
get subcommand, 390
in namespace tcl, 20
list subcommand, 390
platform::generic command, 389
platform::identify command, 388
platform::patterns command, 389
platform::shell::generic command, 389
platform::shell::identify command, 389
platform::shell::platform command, 389
platform::shell package, 389
platform command
in namespace platform::shell, 389
platform package, 388-390
POSIX seconds, 215
pow command, 156
precedence
of operators, 152-153
precision
floating point conversion, 86
format command specifier, 86
prefix command (see tcl::prefix)
in namespace tcl, 98
print_args command, 617
print_array command, 618
print_dict command, 617
print_list command, 617
print_sorted command, 617
private
forwards, 412-413
methods, 412-413
variables, 413-414
private command, 411
private contexts, 410-414
proc command, 39, 39
(see also procedures)
qualified names, 348
within namespaces, 348
procedure bodies
introspection, 46
procedure evaluation
unwinding call stack, 321
procedures, 39-46, 39
(see also proc command)
anonymous, 43-45
body, 39
defining, 39
enumerating, 45
introspection, 45-46
named, 39
parameters, 40-42
reconstructing, 46
returning results, 43

processes
background, 474
identifiers
in pipelines, 474, 479
monitoring, 484-486
pipeline, 463
starting, 463-474
process identifier, 19
profiler::init command, 611
profiler package, 611
profiles (see encoding profiles)
profiling, 611-612
program arguments, 17
properties
introspection
enumeration, 447
property command, 436, 437
prototype-based programming, 430
punycode
punycode encoding, 195
put method
zlib streams, 176, 176, 178, 179
puts (see chan command, puts)
puts command
non-blocking mode, 492
pwd command, 18

Q

q
binary format specifier, 166
binary scan specifier, 171
Q
binary format specifier, 166
quantifiers
controlling greediness in regular expressions, 210
in regular expressions, 202-203
quoting
braces vs. double quotes, 30
definition, 28-31
using braces, 29
using double quotes, 28

R

r
binary format specifier, 165
binary scan specifier, 170
R
binary format specifier, 165
binary scan specifier, 170
rand command, 156
RC4 cipher, 500
read (see chan command, read)
readable
vwait -extended value, 453
readable event (see events, readable)
read eval print loop, 277
readFile command, 259
recursion
tailcall command, 282-286
reflected channels
blocking subcommand, 510
cgetall subcommand, 509
cget subcommand, 509
configure subcommand, 509
creating, 514-515
data input, 514-514
data output, 512-514
event-driven I/O, 510-512
finalize subcommand, 509
initialize subcommand, 508
limitations, 519-520
non-blocking I/O, 510-512
read subcommand, 514-514
seeking, 515-516
watch subcommand, 510
write subcommand, 512-514
reflected channels, seek subcommand, 515
regexp command, 198-212, 198
(see also regular expressions)
embedded options, 208
ignoring character case, 208
newline-sensitive matching, 209
retrieving matched subexpressions, 206
register command
in namespace http, 528
registry package, 615
regsub command, 212-214
-all option, 212
regular expressions, 197, 197, 197
(see also matching with regexp)
(see also substitutions with regsub)
alternation, 203-203
anchors, 203-203
atoms, 201-202
back references, 205-206
bracket expressions, 199
branches, 203-203
capturing parenthesis, 201
character classes, 199, 199
character escapes, 198
character ranges, 199
constraint escapes, 204
constraints, 203-205
counting matches, 206
embedded comments, 211
embedded options, 208-208
escaping metacharacters, 209
expanded syntax, 211
for searching lists, 120
greediness, 210
groups, 201-202, 201
ignoring whitespace, 211
lookahead constraints, 204
matching any character, 198
matching at an offset, 210
matching literal characters, 198
matching literals, 208-209
newline-sensitivity, 209-209
non-capturing parenthesis, 202
quantifiers, 202-203
retrieving all matches, 207
retrieving matches, 206-207
retrieving matches inline, 207
retrieving match indices, 207
selecting array elements, 54
substituting, 212-214
syntax, 197-197
rename command, 35
for command deletion, 36
renamemethod command, 400
renaming
objects, 441
REPL, 10, 277
implementation, 496
replace profile, 182
replacing
strings, 77
reset method
zlib streams, 177
resource management
in coroutines, 571
using traces, 295-296, 296
using try..finally, 331
return
exceptional, 316
normal, 316
return codes, 314-316
break, 315, 317
continue, 315, 316
custom, 325
error, 315
generating, 319-325
ok, 315
predefined, 315
propagation, 316-319
return, 315, 317
return command, 43-43, 319-325
as identity function, 321
custom options, 325
emulating break, 324
implementing control statements, 336
raising errors, 333-334
return options dictionary, 319, 327-328
creating, 320
RFC 1950, 173
RFC 1951, 172, 173
RFC 1952, 174
RFC 2822, 227 (see time, RFC 2822 format)
roots command
in namespace tcl::tm, 386
round command, 156

S

s
binary format specifier, 165
binary scan specifier, 170
S
binary format specifier, 165
binary scan specifier, 170
safe::interpAddToAccessPath command, 549
safe::interpConfigure command, 548
safe::interpCreate command, 547, 549
safe::interpDelete command, 547
safe::interpFindInAccessPath command, 549
safe::interpInit command, 547, 549
safe::setLogCmd command, 549
safe::setSyncMode command, 549
safe interpreters, 540-550
aliases, 541-542
creating, 540-541
safe namespace, 546-550
Safe Tcl, 546-550
sync mode, 549
scan command, 87-93
floating point specifiers, 89
integer specifiers, 89
maximum field widths, 92
string specifiers, 89
XPG3 position specifiers, 91
scheduling scripts, 456-457
scripts, 23
constructing, 65-67
executing with eval, 65-67
loading from files, 68-70
marking as executable on Unix, 15-16
marking as executable on Windows, 16-16
searching
grouped lists, 122-122
nested lists, 121-122
strings, 80
seek (see chan command, seek)
self command, 416, 440
caller subcommand, 445
call subcommand, 427, 445
class subcommand, 442
filter subcommand, 446
method subcommand, 445
namespace subcommand, 440
next subcommand, 427, 445
object subcommand, 440
target subcommand, 445
separators
for splitting strings, 110
serial ports
BREAK condition, 530

CTS, 530
data bits, 529
DSR, 530
DTR, 530
errors, 530-530
flow control, 529
handshake, 529
interfacing to, 529-530
parity, 529
RTS, 530
speed, 529
stop bits, 529
XON/XOFF flow control, 530
set command, 47
setLogCmd command
in namespace safe, 549
setSyncMode command
in namespace safe, 549
SFE (see single file executables)
shared libraries
loading, 382-384
ShiftJIS, 181
short circuit
in expressions, 152, 153
sin command, 156
single file deployment, 603-605
singletons, 435-436
sinh command, 156
size modifier, 92
sleeping, 456-456
Snit, 391
socket command
clients, 521-523
asynchronous connects, 522-523
synchronous connects, 522-522
server, 524-526
sorting, 115
(see also lsort command)
sorting lists, 115-119
source
building, 6-9
source command, 68-70
space character
format flag, 86
split command, 110-111
splitx command, 110
sqrt command, 156
srand command, 156
stack (see call stack)
startup file
for tclsh, 10
stderr, 249
encoding profile, 263
stdin, 249
stdout, 249
strict profile, 182
string
definition, 73-74
interpolation, 29
string command
cat subcommand, 78
compare subcommand, 94
equal subcommand, 93
first subcommand, 80
index subcommand, 75
insert subcommand, 76
is subcommand, 94
last subcommand, 80
length subcommand, 75
map subcommand, 77
for substring deletion, 77
match subcommand, 96
range subcommand, 75
repeat subcommand, 79
replace subcommand, 76
for substring deletion, 76
reverse subcommand, 80
tolower subcommand, 79
totitle subcommand, 79
toupper subcommand, 79
trimleft subcommand, 77
trimright subcommand, 77
trim subcommand, 77
string construction
literals, 75
repeated strings, 79
strings
appending, 76
binary (see binary strings)
changing case, 79
comparing for equality, 93
comparing lexicographically, 94
compressing, 172-179
computing length, 75
concatenating, 78, 79
construction (see string construction)
defined, 73
deleting substrings, 76, 77
formatting, 82-87
indices, 74
interpolation using subst, 81
joining, 78, 79
literals, 75
locating substrings, 80
matching glob patterns, 96
parsing (see parsing)
replacing substrings, 76, 77
retrieving characters by position, 75
retrieving substrings, 75
reversing, 80
splitting into lists, 110-111
trimming, 77-78
validation, 94-96
subclass, 406
subst command, 81
substitution
with regsub, 212-214
substitutions, 24-28
command, 27
in brace-quoted strings, 29
in double-quoted strings, 29
variable, 26-27, 26
substitutions,
double, 66
superclass command, 406, 407
suspending, 456-456
switch command, 61
matching options, 62-63
sync mode
Safe Tcl, 549
syntax, 23-24
comments, 34-35
system
locale, 218
system identification, 388-390

T

t
binary format specifier, 165
binary scan specifier, 170
tailcall command, 282-286
tan command, 156
tanh command, 156
Tcl
benefits, 1-2
history, 1-1
syntax, 23-24
tcl_endOfWord command, 81
tcl_interactive, 478
tcl_interactive global variable, 12
TCL_LIBRARY, 373
tcl_library global variable, 373, 378, 603
TCL_MAJOR_MINOR_TM_PATH environment variable, 386
tcl_nonwordchars global variable, 80
tcl_patchlevel global variable, 20
TCL_PKG_PREFER_LATEST, 380
tcl_pkgPath global variable, 378
tcl_platform, 389
tcl_platform global variable, 21
tcl_precision, 144
tcl_startOfNextWord command, 81
tcl_startOfPreviousWord command, 81
TCL_TZ, 216
tcl_version global variable, 19
tcl_wordBreakAfter command, 81
tcl_wordBreakBefore command, 81
tcl_wordchars global variable, 80
tcl::build-info command, 20
tcl::idna command
decode subcommand, 195
encode subcommand, 195
puny subcommand, 195
tcl::idna package, 195
tcl::mathop:: commands, 145-149
tcl::pkgconfig command, 20
tcl::prefix command
all subcommand, 98
longest subcommand, 98
match subcommand, 98
tcl::process command
autopurge subcommand, 485
list subcommand, 484
purge subcommand, 485
status subcommand, 484-485
tcl::tm::path command
add subcommand, 386
list subcommand, 385
remove subcommand, 387
tcl::tm::roots command, 386
tcl8 profile, 182
Tcl distributions, 6-6
tclIndex, 374
TCLLIBPATH, 378
Tcl shells
tclsh, 10-12
tkcon, 15-15, 15
wish, 13-14
tclshrc.tcl startup file, 10
tclsh shell, 10-12
-encoding option, 12
entering event loop, 451-453
interactive mode, 10-12, 478
running scripts, 12
using in a pipeline, 476-479
tcltest::cleanupTests command, 609
tcltest::test command, 608
tcltest package, 607-610
TclVFS, 597
TCP
clients, 521-523
servers, 524-526
TDBC, 615
tell (see chan command, tell)
tell command, 265
terminal configuration, 260-260
test command, 607-610
in namespace tcltest, 608
testing
Tcl commands, 607-610
textutil::split package, 110
Thread package, 615
throw command, 332-333
tilde substitution, 230
time
arithmetic, 225-226
formatting, 217-222
locales
extending, 227
localization, 227-227
parse format, 222-223
parse locales, 223
parsing, 222-225

free-form, 224
RFC 2822 format, 227
time command, 614-614
timeleft
vwait -extended value, 453
timerate command, 612-614
timers, 456-457
cancelling, 460
time zones, 216
for formatting times, 218
format, 216
tkcon shell, 15-15, 15
Tk package, 615
TLS protocol, 528
trace command, 288
add command, 296-297
add execution, 297-299
add variable, 288-296
for arrays, 291-293
info, 299-300
operations for command traces, 296-297
operations for execution traces, 297-299
operations on variables, 289-291
remove command, 299
remove execution, 299
remove variable, 299
traces, 288-300, 288
(see also trace command)
deleting, 299
for data flow, 294-295
for lazy initialization, 293
for resource management, 295-296, 296
introspecting, 299-300
introspection, 300
Transmission Control Protocol (see TCP)
trap
try clause, 329, 331
truncating files, 266
try command, 329-332
types
boolean, 141-142
double, 142
floating point, 142
integer, 142-142
numeric, 141-145
type specifiers
in binary format command, 163
TZ, 216

U

u
binary scan command flag, 167
Uhler, 305
Unary operators, 152
unexport command, 398
Unicode, 181
backslash sequences, 25
code points, 73-74
unknown command, 33-34
default handler, 33, 374
overriding, 33
unknown commands
within namespaces, 356-357
unknown method, 398
unset command, 49
for array elements, 55
update command, 454-455
idletasks argument, 454
uplevel command, 276-280
implementing control statements, 278, 336
upvar command, 272-275
UTF-8, 181
uuencode encoding
of binary strings, 161

V

validation, 94
(see also string, validation)
input, 94-96
variable
vwait -extended value, 453
variable command, 347
in class definitions, 394
in derived classes, 408
variable contexts, 269-288
variables
appending to, 76
assigning to, 47-47
check if array, 54
checking existence, 52-52, 52
class, 395-396
constant, 294
creation vs. definition, 50
deleting traces, 299
enumerating, 51-51
enumerating globals, 51
enumerating locals, 51
global (see global variables)
instance, 394-395
introspection
enumeration, 447
lifetime of globals, 50
lifetime of locals, 49
linking locals to globals, 50
linking to variables in other procedures, 272-275
local, 49
linking to namespace variables, 351
name syntax, 46-47
predefined, 59
private, 413-414
referencing with $, 47
referencing with ${}, 47, 48
resolving names (see name resolution)
retrieving values, 47-47
scope, 49
starting a trace, 288-296
substitution, 26-27, 26
traces
for resource management, 296
to detect coroutine termination, 572
unsetting, 49
within namespaces, 347-348
versions
obtaining Tcl version, 19
VFS, 597
vfs package, 615
Virtual File Systems (see VFS)
vwait
conditions, 452
vwait command, 451-453
deadlock, 453

W

w
binary format specifier, 165
binary scan specifier, 170
W
binary format specifier, 165
binary scan specifier, 170, 171
wait command, 618
while command, 63
wide command, 144, 156
wide integers
converting to, 144
wildcard characters, 97-97
wishrc.tcl startup file, 13
wish shell, 13-14
interactive mode, 13-14
running scripts, 14
words, 23
boundaries, 23, 26
bracketed, 24
quoted, 24
words,
boundaries, 28
wrap
commands, 35
writable
vwait -extended value, 453
writeFile command, 259

X

x
binary format specifier, 166
binary scan specifier, 171
X
binary format cursor positioning, 166
binary scan specifier, 171
XoTcl, 391
XPG3 format position specifiers
use for localization, 85
XPG3 position specifiers
format command, 84
scan command, 91

Y

yield command, 565-567
yieldm command, 568
yieldto command, 568-569
for returning exceptions, 572

Z

ZIP archives
creating from directories, 598-598
creating from file lists, 599-599
zip compression, 172
zipfs command, 597
canonical subcommand, 602
exists subcommand, 601
find subcommand, 601
info subcommand, 602
list subcommand, 602
lmkimg subcommand, 604
lmkzip subcommand, 599
mkimg subcommand, 604
mkzip subcommand, 598
mountdata subcommand, 600-600
mount subcommand, 599
introspecting ZIP archives, 601-601
introspecting ZipFS mounts, 600-601
mounting ZIP archives, 599-600
root subcommand, 599
unmount subcommand, 601
zlib command, 172-179
adler32 subcommand, 179
channel transforms, 505-507
compress subcommand, 173
crc32 subcommand, 179
decompress subcommand, 173
deflate subcommand, 173
gunzip subcommand, 174
gzip subcommand, 174
inflate subcommand, 173
push subcommand, 505
stream subcommand, 175, 175
zlib streams, 175-179
closing, 178

compressing, 176-176
creating, 175
decompressing, 178-178
-finalize command, 176
-flush command, 179
flushing, 178-179
-fullflush command, 179
options, 175
reading, 176
reusing, 177
writing, 176