Practical Programming in T cl and Tk

Brent Welch

DRAFT, January 13, 1995

Updated for Tcl 7.4 and Tk 4.0

This book will be published by
Prentice Hall

Please return marked up copy (beforeFebruary 20, 1995) to:

Brent Welch
Xerox-PARC
3333 Coyote Hill Road
Palo Alto CA, 94304

Or, send comments via email to
welch@parc.xerox.com
with the word “book” in the subject.

The book is scheduled for publication in March or April, 1995.

Table of Contents

1. Tcl Fundamentals 1
    - Getting Started 1
    - Tcl Commands 2
    - Hello World 3
    - Variables 3
    - Command Substitution 4
    - Math Expressions 4
    - Backslash Substitution 6
    - Double Quotes 7
    - Procedures 7
    - A While Loop Example 8
    - Grouping And Command Substitution 10
    - More About Variable Substitution 11
    - Substitution And Grouping Summary 11
    - Fine Points 12
    - Comments 13
    - Command Line Arguments 13
    - Reference 14
        - Backslash Sequences 14
        - Arithmetic Operators 14
        - Built-in Math Functions 15
        - Core Tcl Commands 15
        - Predefined Variables 18

2. Strings and Pattern Matching 19
    - The string Command 19
    - Strings And Expresssions 20
    - The append Command 21
    - The format Command 21
    - The scan Command 23
    - String Matching 24
    - Regular Expressions 25
    - The regexp Command 26
    - The regsub Command 28

3. Tcl Data Structures 29
More About Variables 29
The unset command 30
Using info to find out about variables 30
Tcl Lists 31
Constructing Lists: list, lappend, and concat 32
Getting List Elements: llength, lindex, and lrange 33
Modifying Lists: linsert and lreplace 34
Searching Lists: lsearch 34
Sorting Lists: lsort 35
The split And join Commands 35
Arrays 36
The array Command 37
Environment Variables 38
Tracing Variable Values 39

4. Control Flow Commands 41
If Then Else 42
Switch 43
Foreach 44
While 45
For 46
Break And Continue 46
Catch 46
Error 48
Return 49

5. Procedures and Scope 51
The proc Command 51
Changing command names with rename 52
Scope 53
The global Command 53
Use Arrays for Global State 55
Call By Name Using upvar 55
Passing arrays by name 56
The uplevel Command 57

6. Eval 59
Eval And List 59

Eval And Concat 61
Double-quotes and eval 62
Commands That Concat Their Arguments 62
The subst Command 63

7. Working with UNIX 65
Running Unix Programs With exec 65
auto_noexec 67
Looking At The File System 67
Input/Output 70
Opening Files For I/O 70
Reading And Writing 72
The puts and gets commands 72
The read command 73
Random access I/O 73
Closing I/O streams 74
The Current Directory - cd And pwd 74
Matching File Names With glob 74
The exit And pid commands 75

8. Reflection and Debugging 77
The info Command 77
Variables 78
Procedures 79
The call stack 79
Command evaluation 80
Scripts and the library 80
Version numbers 80
Interactive Command History 81
History syntax 82
A comparision to /bin/csh history syntax 82
Debugging 83
Don Libes' debugger 84
Breakpoints by pattern matching 85
Deleting break points 86
The tkerror Command 87
The tkinspect Program 87
Performance Tuning 87

9. Script Libraries 89
The unknown Command 89
The tclIndex File 90
Using A Library: auto_path 90
Disabling the library facility: auto_noload 91
How Auto Loading Works 91
Interactive Conveniences 92
Auto Execute 92
History 92
Abbreviations 92
Tcl Shell Library Environment 93
Coding Style 93
A module prefix for procedure names 94
A global array for state variables 94

10. Tk Fundamentals 95
Hello World In Tk 96
Naming Tk Widgets 98
Configuring Tk Widgets 98
About The Tk Man Pages 99
Summary Of The Tk Commands 99

11. Tk by Example 103
ExecLog 103
Window title 105
A frame for buttons, etc. 105
Command buttons 106
A label and an entry 106
Key bindings and focus 106
A resizable text and scrollbar 107
The Run proc 107
The Log procedure 108
The Stop procedure 108
The Example Browser 109
More about resizing windows 110
Managing global state 111
Searching through files 111
Cascaded menus 112
The Browse proc 112

A Tcl Shell 113
Naming issues 114
Text marks and bindings 114

12. The Pack Geometry Manager 115

Packing towards a side 116
Shrinking frames and pack propagate 116
Horizontal And Vertical Stacking 117
The Cavity Model 118
Packing Space and Display Space 119
The -fill option 119
Internal padding with -ipadx and -ipady 120
External padding with -padx and -pady 123
Expand And Resizing 123
Anchoring 125
Packing Order 126
pack slaves and pack info 127
Pack the scrollbar first 127
Choosing The Parent For Packing 128
Unpacking a Widget 129
Packer Summary 129
The pack Command 130
The Place Geometry Manager 130
The place Command 131
Window Stacking Order 132

13. Binding Commands to X Events 133

The bind Command 133
All, Class, And Widget Bindings 134
The bindtags command 135
break and continue in bindings 135
A note about bindings in earlier versions of Tk ... 135
Event Syntax 136
Key Events 137
Button Events 138
Other Events 138
Modifiers 139
Events in Tk 3.6 and earlier 141
Event Sequences 141

Event Keywords 142

14. Buttons and Menus 145
Button Commands and Scope Issues 145
Buttons Associated with Tcl Variables 149
Button Attributes 151
Button Operations 153
Menus and Menubuttons 153
Manipulating Menus and Menu Entries 155
A Menu by Name Package 156
Popup Menus and Option Menus 159
Keyboard Traversal 159
Menu Attributes 160

15. Simple Tk Widgets 163
Frames and Top-Level Windows 163
Attributes for frames and toplevels 164
The label Widget 165
Label attributes 166
Label width and wrapLength 166
The message Widget 167
Message Attributes 168
Arranging Labels and Messages 169
The scale Widget 169
Scale attributes 170
Programming scales 171
The scrollbar Widget 172
Scrollbar attributes 174
Programming scrollbars 175
The Tk 3.6 protocol 175
The bell Command 176

16. Entry and Listbox Widgets 177
The entry Widget 177
entry attributes 180
Programming entry widgets 181
The listbox Widget 183
Programming listboxes 185
Listbox Bindings 189
Browse select mode 190

Single select mode 190
Extended select mode 191
Multiple select mode 192
Scroll bindings 192
listbox attributes 193
Geometry gridding 194

17. Focus, Grabs, and Dialogs 195

Input Focus 195
The focus command 196
Focus follows mouse 196
Click to type 197
Hybrid models 197
Grabbing the Focus 197
Dialogs 198
The tkwait Command 198
Prompter dialog 198
Destroying widgets 200
Focusing on buttons 200
Animation with the update command 200
File Selection Dialog 201
Creating the dialog 201
Listing the directory 204
Accepting a name 205
Easy stuff 207
File name completion 207

18. The text Widget 211

Text widget taxonomy 211
Text Indices 212
Text Marks 213
Text Tags 214
Tag attributes 215
Mixing attributes from different tags 216
Line Spacing and Justification 217
The Selection 219
Tag Bindings 219
Embedded Widgets 220
Text Bindings 222

Text Operations 223
Text Attributes 225

19. The canvas Widget 227

Hello, World! 227
The Double-Slider Example 229
Canvas Coordinates 233
Arcs 233
Bitmap Items 235
Images 236
Line Items 236
Oval Items 238
Polygon Items 239
Rectangle Items 240
Text Items 241
Window Items 244
Canvas Operations 246
Generating Postscript 248
Canvas Attributes 250
Hints 251
Large coordinate spaces 251
Scaling and Rotation 251
X Resources 252
Objects with many points 252

20. Selections and the Clipboard 253

The selection Command 254
The clipboard Command 255
Interoperation with OpenLook 255
Selection Handlers 255
A canvas selection handler 256

21. Callbacks and Handlers 259

The after Command 259
The fileevent Command 260
The send Command 261
The sender script 262
Using sender 264
Hooking the browser to a shell 266

22. Tk Widget Attributes 269
- Configuring Attributes 269
- Size 270
- Borders and Relief 272
- The Focus Highlight 273
- Padding and Anchors 274
- Putting It All Together 275

23. Color, Images, and Cursors 277
- Colors 278
- Colormaps and Visuals 280
- Bitmaps and Images 281
  - The image Command 281
  - bimap images 281
  - The bitmap attribute 282
  - photo images 283
- The Mouse Cursor 285
- The Text Insert Cursor 287

24. Fonts and Text Attributes 289
- Fonts 289
- Text Layout 292
- Padding and Anchors 293
- Gridding, Resizing, and Geometry 294
- Selection Attributes 295
- A Font Selection Application 295

25. Window Managers and Window Information 303
- The wm Command 303
  - Size, placement, and decoration 304
  - Icons 305
  - Session state 306
  - Miscellaneous 307
- The winfo Command 308
  - Sending commands between applications 308
  - Family relationships 308
  - Size 309
  - Location 310
  - Virtual root window 311

Atoms and IDs 311
Colormaps and visuals 312
The tk Command 313

26. A User Interface to bind 315
A Binding User Interface 315
A Pair of Listboxes Working Together 317
The Editing Interface 319

27. Using X Resources 323
An Introduction To X Resources 323
Warning: order is important! 325
Loading Option Database Files 325
Adding Individual Database Entries 326
Accessing The Database 326
User Defined Buttons 327
User Defined Menus 328

28. Managing User Preferences 331
App-Defaults Files 331
Defining Preferences 333
The Preferences User Interface 335
Managing The Preferences File 338
Tracing Changes To Preference Variables 340

29. C Programming and Tcl 341
Using the Tcl C Library 342
Application Structure 342
Tcl_Main and Tcl_AppInit 343
The standard main in Tcl 7.3 344
A C Command Procedure 345
Managing The Result's Storage 346
Invoking Scripts From C 347
Bypassing Tcl_Eval 347
Putting A Tcl Program Together 349
An Overview of the Tcl C library 349
Application initialization 350
Creating and deleting interpreters 350
Creating and deleteing commands 350
Managing the result string 350

Lists and command parsing 350
Command pipelines 351
Tracing the actions of the Tcl interpreter 351
Evalutating Tcl commands 351
Manipulating Tcl variables 352
Evalutating expressions 352
Converting numbers 352
Hash tables 352
Dynamic Strings 353
Regular expressions and string matching 353
Tilde Substitution 353
Working with signals 353

30. C Programming and Tk 355

Tk_Main and Tcl_AppInit 355
A Custom Main Program 357
A Custom Event Loop 360
An Overview of the Tk C library. 361
Parsing command line arguments 361
The standard application setup 362
Creating windows 362
Application name for send 362
Configuring windows 362
Window coordinates 362
Window stacking order 363
Window information 363
Configuring widget attributes 363
Safe handling of the widget data structure 363
The selection and clipboard 363
Event bindings 364
Event loop interface 364
Handling X events 364
File handlers 364
Timer events 365
Idle callbacks 365
Sleeping 365
Reporting script errors 365
Handling X protocol errors 365
Using the X resource database. 365

Managing bitmaps 366
Creating new image types 366
Using an image in a widget 366
Photo image types 366
Canvas object support 366
Geometry managment 367
String identifiers (UIDS) 367
Colors and Colormaps 367
3D Borders 368
Mouse cursors 368
Font structures 368
Graphics Contexts 368
Allocate a pixmap 368
Screen measurements 368
Relief style 369
Text anchor positions 369
Line cap styles 369
Line join styles 369
Text justification styles 369
Atoms 369
X resource ID management 369

31. Writing a Tk Widget in C 371
Implementing a New Widget 371
The Widget Data Structure 372
Specifying Widget Attributes 373
The Widget Class Command 375
Widget Instance Command 376
Configuring And Reconfiguring Attributes 378
Displaying The Clock 380
The Window Event Procedure 383
Final Cleanup 384

32. Tcl Extension Packages 387
Extended Tcl 388
Adding tclX to your application 388
More UNIX system calls 389
File operations 389
New loop constructs 389

Command line addons 389
Debugging and development support 389
TCP/IP access 390
File scanning (i.e., awk) 390
Math functions as commands 390
List operations 390
Keyed list data structure 390
String utilities 391
XPG/3 message catalog 391
Memory debugging 391
Expect: Controlling Interactive Programs 391
The core expect commandsl 392
Pattern matching 393
Important variables 393
An example expect script 394
Debugging expect scripts 395
Expect's Tcl debugger 395
The Dbg C interface 396
Handling SIGINT 397
BLT 398
Drag and drop 398
Hypertext 399
Graphs 399
Table geometry manager 399
Bitmap support 399
Background exec 399
Busy window 399
Tracing Tcl commands 399
The old-fashioned cutbuffer 400
Tcl-DP 400
Remote Procedure Call 400
Connection setup 401
Sending network data 401
Using UDP 401
Event processing 401
Replicated objects 402
The [incr tcl] Object System 402
Tcl_AppInit With Extensions 404
Other Extensions 407

Tcl applications 407

33. Porting to Tk 4.0 409
wish 409
Obsolete Features 409
The cget Operation 410
Input Focus Highlight 410
Bindings 410
Scrollbar Interface 411
Pack info 411
Focus 411
Send 412
Internal Button Padding 412
Radio Buttons 412
Entry Widget 412
Menus 413
Listboxes 413
No geometry Attribute 413
Text Widget 413
Canvas scrollincrement 414
The Selection 414
Color Attributes 414
The bell Command 415

List of Examples

1.1 The “Hello, World!” example. 3
1.2 Tcl variables. 3
1.3 Command substitution. 4
1.4 Simple arithmetic...5
1.5 Nested commands..5
1.6 Built-in math functions...5
1.7 Controlling precision with tcl_precision. 5
1.8 Quoting special characters with backslash.................................6
1.9 Continuing long lines with backslashes......................................6
1.10 Grouping with double quotes allows substitutions. 7
1.11 Defining a procedure..8
1.12 A loop that multiplies the numbers from 1 to 10. 9
1.13 Embedded command and variable substitution........................10
1.14 Embedded variable references 11

2.1 Comparing strings. 21
2.2 Regular expression to parse the `DISPLAY` environment variable. ..27

3.1 Using set to return a variable value..30
3.2 Using `info` to determine if a variable exists. 30
3.3 Constructing a list with the `list` command. 32
3.4 Using `lappend` to add elements to a list. 32
3.5 Using concat to splice together lists. 33
3.6 Double quotes compared to the `list` command. 33
3.7 Modifying lists with `linsert` and `lreplace`..............................34
3.8 Deleting a list element by value..34
3.9 Sorting a list using a comparison function. 35
3.10 Use split to turn input data into Tcl lists..................................35
3.11 Using arrays. 36
3.12 What if the name of the array is in a variable. 37
3.13 Converting from an array to a list...38
3.14 `printenv` prints the environment variable values. 39
3.15 Tracing variables...40
3.16 Creating array elements with array traces................................40

4.1 A conditional `if-then-else` command. 42
4.2 Chained conditional with `elseif`. 42
4.3 Using `switch` for an exact match. 43
4.4 Using `switch` with substitutions in the patterns. 44
4.5 Using `switch` with all pattern body pairs grouped with quotes. ..44
4.6 Looping with `foreach`. 44
4.7 Parsing command line arguments. 45
4.8 Using `list` with `foreach`..45

4.9 A `while` loop to read standard input. 46
4.10 A `for` loop. 46
4.11 A standard `catch` phrase. 47
4.12 A longer `catch` phrase. 47
4.13 The results of `error` with no info argument. 48
4.14 Preserving `errorInfo` when calling `error`. 48
4.15 Specifying `errorinfo` with `return`. 49

5.1 Default parameter values. 52
5.2 Variable number of arguments. 52
5.3 Variable scope and Tcl procedures. 53
5.4 A random number generator. 54
5.5 Using arrays for global state. 55
5.6 Print by name. 56
5.7 Improved `incr` procedure. 56
5.8 Using an array to implement a stack. 56

6.1 Using `list` to construct commands. 60
6.2 Using `eval` with `$args`. 61

7.1 Using `exec` on a process pipeline. 66
7.2 A procedure to compare file modify times. 68
7.3 Creating a directory recusively. 69
7.4 Determining if pathnames reference the same file. 69
7.5 Opening a file for writing. 70
7.6 Opening a file using the `POSIX` access flags. 71
7.7 A more careful use of `open`. 71
7.8 Opening a process pipeline. 72
7.9 Prompting for input. 72
7.10 A read loop using `gets`. 73
7.11 A read loop using `read` and `split`. 73
7.12 Finding a file by name. 74

8.1 Printing a procedure definition. 79
8.2 Getting a trace of the Tcl call stack. 80
8.3 Interactive `history` usage. 82
8.4 Implementing special history syntax. 83
8.5 A Debug procedure. 83

9.1 Maintaining a `tclIndex` file. 90
9.2 Loading a `tclIndex` file. 91

10.1 “Hello, World!” Tk program. 96

11.1 Logging the output of a `UNIX` program. 104
11.2 A browser for the code examples in the book. 109
11.3 A Tcl shell in a text widget. 113

12.1 Two frames packed inside the main frame. 116
12.2 Turning off geometry propagation. 116
12.3 A horizontal stack inside a vertical stack. 117
12.4 Even more nesting of horizontal and vertical stacks. 117
12.5 Mixing bottom and right packing sides. 118
12.6 Filling the display into extra packing space. 119
12.7 Using horizontal fill in a menubar. 120
12.8 The effects of internal padding (-ipady). 122
12.9 Button padding vs. packer padding. 122
12.10 The look of a default button. 123
12.11 Resizing without the expand option. 124
12.12 Resizing with expand turned on. 124
12.13 More than one expanding widget. 125
12.14 Setup for anchor experiments. 125
12.15 The effects of non-center anchors. 126
12.16 Animating the packing anchors 126
12.17 Controlling the packing order. 127
12.18 Packing into other relatives. 128

13.1 The binding hierarchy. 134
13.2 Output from the UNIX *xmodmap* program. 140
13.3 Emacs-like binding convention for Meta and Escape. 141

14.1 A troublesome `button` command. 146
14.2 Fixing up the troublesome situation. 147
14.3 A `button` associated with a Tcl procedure. 148
14.4 Radio and Check buttons. 150
14.5 Acommand on a `radiobutton` or `checkbutton`. 151
14.6 A `menu` sampler. 154
14.7 A simple menu-by-name package. 156
14.8 Adding menu entries. 157
14.9 A wrapper for cascade entries. 158
14.10 Using the basic menu package. 158
14.11 Keeping the accelerator display up-to-date. 158

15.1 A `label` that displays different strings. 165
15.2 The `message` widget formats long lines of text. 167
15.3 Controlling the text layout in a message widget. 168
15.4 A `scale` widget. 169
15.5 A `text` widget and two scrollbars. 173

16.1 A command, a `label` and an `entry`. 179
16.2 A `listbox` with scrollbars. 183
16.3 A `listbox` with scrollbars and better alignment. 184
16.4 Choosing items from a `listbox` 188

17.1 Setting focus-follows-mouse input focus model. 196

17.2 A simple dialog. ... 199
17.3 A feedback procedure. ... 200
17.4 A file selection dialog. ... 201
17.5 Listing a directory for fileselect. ... 204
17.6 Accepting a file name. ... 206
17.7 Simple support routines. ... 207
17.8 File name completion. ... 208

18.1 Tag configurations for basic character styles. ... 216
18.2 Line spacing and justification in the `text` widget. ... 217
18.3 An active `text` button. ... 219
18.4 Delayed creation of embedded widgets. ... 221

19.1 The `canvas` Hello, World! example. ... 227
19.2 A double slider `canvas` example. ... 229
19.3 Moving the markers for the double-slider. ... 231
19.4 A large scrollable `canvas`. ... 233
19.5 Canvas `arc` items. ... 234
19.6 `C`anvas `bitmap` items. ... 235
19.7 `C`anvas `image` items. ... 236
19.8 A `canvas` stroke drawing example. ... 237
19.9 Canvas `oval` items ... 238
19.10 Canvas `polygon` items. ... 239
19.11 Dragging out a box. ... 240
19.12 Simple edit bindings for canvas text items. ... 242
19.13 Using a `canvas` to scroll a set of widgets. ... 244
19.14 Generating postscript from a `canvas`. ... 249

20.1 Paste the `PRIMARY` or `CLIPBOARD` selection. ... 253
20.2 A selection handler for `canvas` widgets. ... 256

21.1 A read event file handler. ... 261
21.2 The sender application. ... 262
21.3 Using the sender application. ... 264
21.4 Hooking the browser to an `eval` server. ... 266
21.5 Making the shell into an `eval` server. ... 267

22.1 Equal-sized labels ... 272
22.2 3D relief sampler. ... 273
22.3 Borders and padding. ... 275

23.1 Resources for reverse video. ... 277
23.2 Computing a darker color ... 279
23.3 Specifying an image attribute for a widget. ... 281
23.4 Specifying a bitmap for a widget. ... 282
23.5 The built-in bitmaps ... 282

24.1 FindFont matches an existing font. 291
24.2 Handling missing font errors. 291
24.3 FontWidget protects against font errors. 292
24.4 A gridded, resizable listbox. 295
24.5 A font selection application. 295
24.6 Menus for each font component. 296
24.7 Using variable traces to fix things up. 297
24.8 Listing available fonts. 297
24.9 Determining possible font components. 298
24.10 Creating the `radiobutton` menu entries. 298
24.11 Setting up the `label` and `message` widgets. 299
24.12 The font selection procedures. 300

25.1 Gridded geometry for a `canvas`. 304
25.2 Telling other applications what your name is. 308

26.1 A user interface to widget bindings. 316
26.2 Bind_Display presents the bindings for a given widget or class. 317
26.3 Related listboxes are configured to select items together. 318
26.4 Controlling a pair of listboxes with one scrollbar. 318
26.5 Drag-scrolling a pair of listboxes together. 319
26.6 An interface to define bindings. 320
26.7 Defining and saving bindings. 321

27.1 Reading an option database file. 325
27.2 A file containing resource specifications. 325
27.3 Using resources to specify user-defined buttons. 327
27.4 Defining buttons from the resource database. 328
27.5 Specifying menu entries via resources. 328
27.6 Defining menus from resource specifications. 330

28.1 Preferences initialization. 332
28.2 Adding preference items. 333
28.3 Setting preference variables. 334
28.4 Using the preferences package. 334
28.5 A user interface to the preference items. 335
28.6 Interface objects for different preference types. 336
28.7 Displaying the help text for an item. 338
28.8 Saving preferences settings to a file. 338
28.9 Read settings from the preferences file. 339
28.10 Tracing a Tcl variable in a preference item. 340

29.1 A canonical Tcl main program and Tcl_AppInit. 343
29.2 The `RandomCmd` C command procedure. 345
29.3 Calling C command procedure directly. 348
29.4 A Makefile for a simple Tcl C program. 349

30.1 A canonical Tk main program and Tcl_AppInit. 356
30.2 A custom Tk main program. 357
30.3 Using `Tk_DoOneEvent` with `TK_DONT_WAIT`. 361

31.1 The `Clock` widget data structure. 372
31.2 Configuration specs for the clock widget. 373
31.3 The `ClockCmd` command procedure. 375
31.4 The `ClockInstanceCmd` command procedure. 377
31.5 `ClockConfigure` allocates resources for the widget. 378
31.6 `ComputeGeometry` figures out how big the widget is. 380
31.7 The `ClockDisplay` procedure. 381
31.8 The `ClockEventProc` handles window events. 383
31.9 The `ClockDestroy` cleanup procedure. 384

32.1 A sample expect script. 394
32.2 A SIGINT handler. 397
32.3 Summary of [incr tcl] commands. 403
32.4 Tcl_AppInit and extension packages. 404
32.5 Makefile for supertcl. 406

List of T ables

1-1 Backslash sequences. 14
1-2 Arithmetic Operators from highest to lowest precedence. 14
1-3 Built-in Math functions 15
1-4 Built-in Tcl Commands 15
1-5 Variables defined by *tclsh*. 18

2-1 The `string` command 20
2-2 Format conversions 22
2-3 format flags 22
2-4 Regular Expression Syntax 25

3-1 List-related commands 31
3-2 The array command 37

7-1 Summary of the `exec` syntax for I/O redirection. 66
7-2 The Tcl `file` command options. 67
7-3 Tcl commands used for file access. 70
7-4 Summary of the `open` access arguments. 71
7-5 Summary of `POSIX` flags for the access argument. 71

8-1 The `info` command. 78
8-2 The `history` command. 81
8-3 Special `history` syntax. 82
8-4 Debugger commands. 85

10-1 Tk widget-creation commands 100
10-2 Tk widget-manipulation commands 100

12-1 A summary of the `pack` command. 130
12-2 Packing options. 130
12-3 A summary of the `place` command. 131
12-4 Placement options. 132

13-1 Event types. Comma-separated types are equivalent. 136
13-2 Event modifiers. 139
13-3 A summary of the event keywords. 142

14-1 Resource names of attributes for all `button` widgets. 152
14-2 Button operations. 153
14-3 Menu entry index keywords 155
14-4 Menu operations. 155
14-5 Resource names of attributes for `menu` widgets 160
14-6 Attributes for menu entries. 161

15-1 Resource names of attributes for `frame` and `toplevel` widgets. 164
15-2 Resource names of attributes for `label` widgets. 166
15-3 Resource names for attributes for `message` widgets. 168
15-4 Default bindings for `scale` widgets. 170
15-5 Resource names for attributes for `scale` widgets. 170
15-6 Operations on `scale` widgets.. 171
15-7 Default bindings for `scrollbar` widgets. 174
15-8 Resource names of attributes for `scrollbar` widgets. 174
15-9 Operations on scrollbar widgets. 175

16-1 Default bindings for `entry` widgets. 178
16-2 Resource names for attributes of `entry` widgets. 180
16-3 Indices for `entry` widgets 181
16-4 Operations on `entry` widgets. 182
16-5 Indices for `listbox` widgets 186
16-6 Operations on `listbox` widgets. 186
16-7 The values for the `selectMode` of a `listbox`. 190
16-8 Bindings for `browse` selection mode. 190
16-9 Bindings for a `listbox` in `single` selectMode. 190
16-10 Bindings for `extended` selection mode. 191
16-11 Bindings for `multiple` selection mode. 192
16-12 Scroll bindings common to all selection modes. 193
16-13 Resource names of attributes for `listbox` widgets. 193

17-1 The `focus` command. 196
17-2 The `grab` command. 197
17-3 The `tkwait` command. 198

18-1 Forms for the indices in `text` widgets. 212
18-2 Index modifiers for `text` widgets. 213
18-3 Attributes for `text` tags. 215
18-4 Options to the `window create` operation. 221
18-5 Bindings for the `text` widget. 222
18-6 Operations for the `text` widget. 224
18-7 Resource names of attributes for `text` widgets. 226

19-1 Attributes for `arc` canvas items. 234
19-2 Attributes for `bitmap` canvas items. 235
19-3 Attributes for `image` canvas items. 236
19-4 Attributes for `line` canvas items. 238
19-5 Attributes for `oval` canvas items. 239
19-6 Attributes for `polygon` canvas items. 240
19-7 Attributes for rectangle canvas items. 241
19-8 Indices for canvas `text` items 241
19-9 Canvas operations that apply to `text` items. 242
19-10 Attributes for `text` canvas items. 244
19-11 Operations on a `canvas` widget. 246

19-12 Canvas postscript options. 248
19-13 Resource names of attributes for the canvas widget. 250

20-1 The selection command. 254
20-2 The clipboard command. 255

21-1 The after command. 260
21-2 The fileevent command. 261

22-1 Size attribute resource names. 270
22-2 Border and relief attribute resource names. 272
22-3 Border and relief attribute resource names. 274
22-4 Layout attribute resource names. 274

23-1 Color attribute resource names. 278
23-2 Visual classes for X displays. Values for the visual attribute. 280
23-3 Summary of the image command. 281
23-4 Bitmap image options 282
23-5 Photo image attributes 283
23-6 Photo image operations. 284
23-7 Image copy options. 285
23-8 Image read options. 285
23-9 Image write options. 285
23-10 Cursor attribute resource names. 287

24-1 X Font specification components. 290
24-2 Resource names for layout attributes. 293
24-3 Resource names for padding and anchors. 293
24-4 Geometry commands affected by gridding. 294

25-1 Size, placement and decoration window manager operations. 305
25-2 Window manager commands for icons. 306
25-3 Session-related window manager operations. 307
25-4 Miscellaneous window manager operations. 307
25-5 Information useful with the send command. 308
25-6 Information about the window hierarchy. 309
25-7 Information about the window size. 310
25-8 Information about the window location. 310
25-9 Information associated with virtual root windows. 311
25-10 Information about atoms and window ids. 312
25-11 Information about colormaps and visual classes. 312

31-1 Configuration flags and corresponding C types. 374

33-1 Changes in color attribute names. 414

Preface

I first heard about Tcl from John Ousterhout in 1988 while I was his Ph.D. student at Berkeley. We were designing a network operating system, Sprite. While the students hacked on a new kernel, John was writing a new editor and terminal emulator. He used Tcl as the command language for both tools so that users could define menus and otherwise customize those programs. This was in the days of X10, and he had plans for an X toolkit based on Tcl that would allow programs to cooperate by communicating with Tcl commands. To me, this cooperation among tools was the essence of the Tool Command Language (Tcl).

That early vision imagined that applications would be large bodies of compiled code and a small amount of Tcl used for configuration and high-level commands. John’s editor, `mx`, and the terminal emulator, `tx`, followed this model. While this model remains valid, it has also turned out to be possible to write entire applications in Tcl. This is because of the Tcl/Tk shell, `wish`, that provides all the functionality of other shell languages, which includes running other programs, plus the ability to create a graphical user interface. For better or worse, it is now common to find applications that contain thousands of lines of Tcl script.

This book came about because, while I found it enjoyable and productive to use Tcl and Tk, there were times when I was frustrated. In addition, working at Xerox PARC, with many experts in languages and systems, I was compelled to understand both the strengths and weaknesses of Tcl and Tk. While many of my colleagues adopted Tcl and Tk for their projects, they were also just as quick to point out its flaws. In response, I have built up a set of programming techniques that exploit the power of Tcl and Tk while avoiding troublesome areas. Thus, this book is meant as a practical guide that will help you get the most out of Tcl and Tk while avoiding some of the frustrations that I experienced.

Who Should Read This Book

This books is meant to be useful to the beginner as well as the expert in Tcl. For the beginner and expert alike I recommend careful study of the first chapter on Tcl. The programming model of Tcl is different from many programming languages. The model is based on string substitutions, and it is important that you understand it properly to avoid trouble later on. The remainder of the book consists of examples that should help you get started using Tcl and Tk productively.

This book assumes that you have some UNIX and X background, although you should be able to get by even if you are a complete novice. Expertise in UNIX shell programming will help, but it is not required. Where aspects of X are relevant, I will try to provide some background information.

How To Read This Book

This book is best used in a hands-on manner, at the computer trying out the examples. The book tries to fill the gap between the terse Tcl and Tk manual pages, which are complete but lack context and examples, and existing Tcl programs that may or may not be documented or well written.

I recommend the on-line manual pages for the Tcl and Tk commands. They provide a detailed reference guide to each command. This book summarises some of the information from the man pages, but it does not provide the complete details, which can vary from release to release.

I also recommend the book by Ousterhout, *Tcl and the Tk Toolkit*, which provides a broad overview of all aspects of Tcl and Tk. There is some overlap with Ousterhout's book, although that book provides a more detailed treatment of C programming and Tcl.

How To Review This Book

At this point I am primarily concerned with technical issues. Don't worry too much about spelling and other copy edits. Concentrate on the examples and passages that are confusing. You can mark up the manuscript with a pen and return it to me. Or, send me email at welch@parc.xerox.com with the subject "tcl book". This is the last major draft I will post before getting the book ready for final publication. If you can return your comments by mid to late February it would be best. Thanks, in advance!

I would like to highlight a few key spots in the manuscripts as "hot tips". If you could nominate one or more such paragraphs from each chapter I will add some sort of icon to the margin to indicate the "reviewer-selected" hot tips!

Thanks

Many thanks to the patient reviewers of early drafts: Don Libes, Dan Swinehart, Carl Hauser, Pierre David, Jim Thornton, John Maxwell, Hador Shemtov, Charles Thayer, Ken Pier. [UPDATE] (Mention email reviews, too)

Introduction

This introduction gives an overview of Tcl and the organization of this book.

Why Tcl? Is it just another shell language? How can it help you?

Tcl stands for *Tool Command Language*. Tcl is really two things: a scripting language, and an interpreter for that language that is designed to be easy to embed into your application. Tcl and its associated X windows toolkit, Tk, were designed and crafted by Prof. John Ousterhout of U.C. Berkeley. These packages can be picked up off the Internet (see below) and used in your application, even a commercial one. The interpreter has been ported from UNIX to DOS and Macintosh environments.

As a scripting language, Tcl is similar to other UNIX shell languages such as the Bourne Shell, the C Shell, the Korn Shell, and Perl. Shell programs let you execute other programs. They provide enough programmability (variables, control flow, procedures) that you can build up complex scripts that assemble existing programs into a new tool tailored for your needs. Shells are wonderful for automating routine chores.

It is the ability to easily add a Tcl interpreter to your application that sets it apart from other shells. Tcl fills the role of an extension language that is used to configure and customize applications. There is no need to invent a command language for your new application, or struggle to provide some sort of user-programmability for your tool. Instead, by adding a Tcl interpreter you are encouraged to structure your application as a set of primitive operations that can be composed by a script to best suit the needs of your users. It also allows programmatic control over your application by other programs, leading to suites of applications that work together well.

There are other choices for extension languages that include Scheme, Elisp, and Python. Your choice between them is partly a matter of taste. Tcl has simple constructs and looks somewhat like C. It is also easy to add new Tcl primitives by writing C procedures. By now there are a large number of Tcl commands that have been contributed by the Tcl community. So another reason to choose Tcl is because of what you can access from Tcl scripts "out-of-the-box". To me, this is more important than the details of the language.

The Tcl C library has clean interfaces and is simple to use. The library implements the basic interpreter and a set of core scripting commands that implement variables, flow control, file I/O, and procedures (see page 15). In addition, your application can define new Tcl commands. These commands are associated with a C or C++ procedure that your application provides. The result is that applications are split into a set of primitives written in a compiled language and exported as Tcl commands. A Tcl script is used to compose the primitives into the overall application. The script layer has access to shell-like capability to run other programs and access the file system, as well as call directly into the application by using the application-specific Tcl commands you define. In addition, from the C programming level, you can call Tcl scripts, set and query Tcl variables, and even trace the execution of the Tcl interpreter.

There are many Tcl extensions freely available on the net. Most extensions include a C library that provides some new functionality, and a Tcl interface to the library. Examples include socket access for network programming, database access, telephone control, MIDI controller access, and `expect`, which adds Tcl commands to control interactive programs.

The most notable extension is Tk, a toolkit for X windows. Tk defines Tcl commands that let you create and manipulate user interface widgets. The script-based approach to UI programming has three benefits. First, development is fast because of the rapid turnaround - there is no waiting for long compilations. Second, the Tcl commands provide a higher-level interface to X than most standard C library toolkits. Simple interfaces require just a handful of commands to define them. At the same time, it is possible to refine the interface in order to get every detail just so. The fast turnaround aids the refinement process. The third advantage is that the user interface is clearly factored out from the rest of your application. The developer can concentrate on the implementation of the application core, and then fairly painlessly work up a user interface. The core set of Tk widgets is often sufficient for all your UI needs. However, it is also possible to write custom Tk widgets in C, and again there are many contributed Tk widgets available on the network.

Ftp Archives

The network archive site for Tcl is `ftp.aud.alcatel.com`. Under the `/tcl` directory there are subdirectories for the core Tcl distributions (`sprite-mirror`, for historical reasons), contributed extensions (`extensions`), contributed applications (`code`), documentation (`docs`), and Tcl for non-UNIX platforms (`distrib`). Mirror sites for the archive include:

```
ftp://syd.dit.csiro.au/pub/tk/contrib
ftp://syd.dit.csiro.au/pub/tk/sprite
```

```
ftp://ftp.ibp.fr/pub/tcl/distrib
ftp://ftp.ibp.fr/pub/tcl/contrib
ftp://ftp.ibp.fr/pub/tcl/expect
     (ftp.ibp.fr = 132.227.60.2)
ftp://src.doc.ic.ac.uk/packages/tcl/tcl-archive
     (src.doc.ic.ac.uk = 146.169.2.10)
ftp://ftp.luth.se/pub/languages/tcl/
http://ftp.luth.se/pub/langugages/tcl/
ftp://ftp.switch.ch/mirror/tcl
     (Contact address: switchinfo@switch.ch)
ftp://ftp.sterling.com/programming/languages/tcl
ftp://fpt.sunet.se/pub/lang/tcl
mailto://ftpmail@ftp.sunet.se
     (Contact: archive@ftp.sunet.se)
ftp://ftp.cs.columbia.edu/archives/tcl
     (Contact: ftp@cs.columbia.edu)
ftp://ftp.uni-paderborn.de/pub/unix/tcl/alcatel
ftp://sunsite.unc.edu/pub/languages/tcl/
ftp://iskut.ucs.ubc.ca/pub/X11/tcl/
ftp://ftp.funet.fi/pub/languages/tcl/
ftp://coma.cs.tu-berlin.de/pub/tcl/
ftp://nic.funet.fi/pub/languages/tcl/
```

You can verify the location of the Tcl archive by using the `archie` service to look for sites that contain the Tcl distributions. Archie is a service that indexes the contents of anonymous FTP servers. Information about using `archie` can be obtained by sending mail to `archie@archie.sura.net` that contains the message `Help`.

World Wide Web

There are a number of Tcl pages on the world-wide-web:

```
http://www.sco.com/IXI/of_interest/tcl/Tcl.html
http://web.cs.ualberta.ca/~wade/Auto/Tcl.html
```

Typographic Conventions

The more important examples are set apart with a title and horizontal rules, while others appear in-line as shown below. The examples use `courier` for Tcl and C code. When interesting results are returned by a Tcl command, those are presented below in *`oblique courier`*, as shown below. The => is not part of the return value.

```
expr 5 + 8
=> 13
```

The `courier` font is also used when naming Tcl commands and C procedures within sentences.

The usage of a Tcl command is presented as shown below. The command name and constant keywords appear in `courier`. Variable values appear in *courier oblique*. Optional arguments are surrounded with question marks.

```
set varname ?value?
```

The name of a UNIX program is in italics, e.g. *xterm*.

Tcl 7.4 and Tk 4.0

This book is up-to-date with Tcl version 7.4 and Tk version 4.0. There are occasional descriptions of Tk 3.6 features. The last chapter has some notes about porting scripts written in earlier versions of Tk.

Book Organization

The first chapter of this book describes the fundamental mechanisms that characterize the Tcl language. This is an important chapter that provides the basic grounding you will need to use Tcl effectively. Even if you have programmed in Tcl already, you should review this chapter.

Chapters 2-5 cover the basic Tcl commands in more detail, including string handling, regular expressions, data types, control flow, procedures and scoping issues. You can skip these chapters if you already know Tcl.

Chapter 6 discusses `eval` and more advanced Tcl coding techniques. If you are running into quoting problems, check out this chapter.

Chapter 7 describes the interface to UNIX and the shell-like capabilities to run other programs and examine the file system. The I/O commands are described here.

Chapter 8 describes the facilities provided by the interpreter for introspection. You can find out about all the internal state of Tcl. Development aids and debugging are also covered here.

Chapter 9 describes the script library facility. If you do much Tcl programming, you will want to collect useful scripts into a library. This chapter also describes coding conventions to support larger scale programming efforts.

Chapter10 is an introduction to Tk. It explains the relevant aspects of X and the basic model provided by the Tk toolkit.

Chapter 11 illustrates Tk programming with a number of short examples. One of the examples is a browser for the code examples in this book.

Chapter 12 explains geometry management, which is responsible for arranging widgets on the screen. The chapter is primarily about the packer geometry manager, although the simpler place geometry manager is also briefly described.

Chapter 13 covers event binding. A binding registers a Tcl script that is executed in response to events from the X window system.

Chapter 14 describes the `button` and `menu` widgets. The chapter includes a simple menu package that hides some of details of setting up Tk menus.

Chapter 15 describes several simple Tk widgets: the `frame`, the `toplevel`, the `label`, the `message`, the `scale`, and the `scrollbar`. These widgets can be added to your interface with two or three commands. The `bell` command is also covered here.

Chapter 16 describes the `entry` and `listbox` widgets. These are specialized text widgets that provide a single line of text input and a scrollable list of text items, respectively. You are likely to program specialized behavior for these widgets.

Chapter 17 covers the issues related to dialog boxes. This includes input focus and grabs for modal interactions. It includes a file selection dialog box as an example.

Chapter 18 describes the `text` widget. This is a general purpose text widget with advanced features for text formatting, editting, and embedded images.

Chapter 19 describes the `canvas` widget that provides a general drawing interface.

Chapter 20 explains how to use the selection mechanism for cut-and-paste. Tk supports different selections, including the `CLIPBOARD` selection used by OpenLook tools.

Chapter 21 describes the `after`, `fileevent`, and `send` commands. These commands let you create sophisticated application structures, including cooperating suites of applications.

Chapter 22 is the first of three chapters that review the attributes that are shared among the Tk widget set. This chapter describes sizing and borders.

Chapter 23 describes colors, images and cursors. It explains how to use the bitmap and color photo image types. The chapter includes a complete map of the X cursor font.

Chapter 24 describes fonts and other text-related attributes. The extended example is a font selection application.

Chapter 25 explains how to interact with the window manager using the `wm` command. The chapter describes all the information available through the `winfo` command.

Chapter 26 presents a user interface to the binding mechanism. You can browse and edit bindings for widgets and classes with the interface.

Chapter 27 describes the X resource mechanism and how it relates to the Tk toolkit. The extended examples show how end users can use resource specifications to define custom buttons and menus for an application.

Chapter 28 builds upon Chapter 27 to create a user preferences package and an associated user interface. The preference package links a Tcl variable used in your application to an X resource specification.

Chapter 29 provides a short introduction to using Tcl at the C programming level. It gets you started with integrating Tcl into an existing application, and it provides a survey the the facilities in the Tcl C library.

Chapter 30 introduces C programming with the Tk toolkit. It surveys the Tk C library.

Chapter 31 is a sample Tk widget implementation in C. A digital clock widget is built.

Chapter 32 is a survey of several interesting Tcl extension packages. The packages extend Tcl to provide access to more UNIX functionality (TclX), control over interactive programs (Expect), network programming (Tcl-DP), more Tk widgets (BLT), and an object system ([intr tcl]). The chapter concludes with a program that integrates all of these extensions into one *supertcl* application.

Chapter33 has notes about porting your scripts to Tk 4.0.

On-line examples

The final version of this book will include a floppy disk with copies of the examples. In the meantime you will be able to find them via FTP.

```
ftp://parcftp.xerox.com/pub/sprite/welch/examples.tar
```

CHAPTER 1

Tcl Fundamentals

This chapter describes the basic syntax rules for the Tcl scripting language. It describes the basic mechanisms used by the Tcl interpreter: substitution and grouping. It touches lightly on the following Tcl commands: `puts`, `format`, `set`, `expr`, `string`, `while`, `incr`, and `proc`.

Tcl is a string-based command language. The language has only a few fundamental constructs and relatively little syntax, which makes it easy to learn. The basic mechanisms are all related to strings and string substitutions, so it is fairly easy to visualize what is going on in the interpreter. The model is a little different than some other languages you may already be familiar with, so it is worth making sure you understand the basic concepts.

Getting Started

With any Tcl installation there are typically two Tcl shell programs that you can use: `tclsh` and `wish`*. They are simple programs that are not much more than a read-eval-print loop. The first is a basic Tcl shell that can be used as a shell much like the C-shell or Bourne shell. `wish` is a Tcl interpreter that has been extended with the Tk commands used to create and manipulate X widgets. If you cannot find the basic Tcl shell, just run `wish` and ignore for the moment the empty window it pops up. Both shells print a % prompt and will execute Tcl commands interactively, printing the result of each top level command.

You may also find it easier to enter the longer examples into a file using

* You may have variations on these programs that reflect different extensions added to the shells. tcl and wishx are the shells that have Extended Tcl added, for example.

your favorite editor. This lets you quickly try out variations and correct mistakes. Taking this approach you have two options. The first way is to use two windows, one running the Tcl interpreter and the other running your editor. Save your examples to a file and then execute them with the Tcl `source` command.

```
source filename
```

The second way is to create a stand-alone script much like an `sh` or `csh` script. The trick is in the first line of the file, which names the interpreter for the rest of the file. Support for this is built into the `exec` system call in UNIX. Begin the file with either of the following lines.

```
#!/usr/local/bin/tcl
```

or

```
#!/usr/local/bin/wish
```

Of course, the actual pathname for these programs may be different on your system*. Also, on most UNIX systems this pathname is limited to 32 characters, including the `#!`. The 32-character limit is a limitation of the UNIX `exec` system call. If you get the pathname wrong, you get a confusing "command not found" error, and if the pathname is too long you may end up with `/bin/sh` trying to interpret your script, giving you syntax errors.

If you have Tk version 3.6, its version of wish requires a `-f` argument to make it read the contents of a file. The `-f` switch is ignored in Tk 4.0.

```
#!/usr/local/bin/wish -f
```

Tcl Commands

The basic syntax for a Tcl command is:

```
command arg1 arg2 arg3 ...
```

The `command` is either the name of a built-in command or a Tcl procedure. White space is used to separate the command name and its arguments, and a newline or semicolon is used to terminate a command.

The arguments to a command are string-valued. Except for the substitutions described below, the Tcl interpreter does no interpretation of the arguments to a command. This is just the opposite of a language like Lisp in which all identifiers are bound to a value, and you have to explicitly quote things to get strings. In Tcl, everything is a string, and you have to explicitly ask for evaluation of variables and nested commands.

This basic model is extended with just a few pieces of syntax for *grouping*, which allows multiple words in one argument, and *substitution*, which is used with programming variables and nested command calls. The grouping and substitutions are the only mechanisms employed by the Tcl interpreter before it runs a command.

* At Xerox PARC, for example, the pathnames are /import/tcl7/bin/tclsh and /import/tcl7/bin/wish.

Hello World

Example 1–1 The “Hello, World!” example.

```
puts stdout {Hello, World!}
=> Hello, World!
```

In this example the command is `puts`, which takes two arguments: an I/O stream identifier and a string. `puts` writes the string to the I/O stream along with a trailing newline character. There are two points to emphasize:

- Arguments are interpreted by the command. In the example, `stdout` is used to identify the standard output stream. The use of `stdout` as a name is a convention employed by `puts` and the other I/O commands. Also, `stderr` is used to identify the standard error output, and `stdin` is used to identify the standard input.
- Curly braces are used to group words together into a single argument. The braces get stripped off by the interpreter and are not part of the argument. The `puts` command receives `Hello, World!` as its second argument.

Variables

The `set` command is used to assign a value to a variable. It takes two arguments: the first is the name of the variable and the second is the value. Variable names can be any length, and case is significant. It is not necessary to declare Tcl variables before you use them. The interpreter will create the variable when it is first assigned a value. The value of a variable is obtained later with the dollar-sign syntax illustrated below.

Example 1–2 Tcl variables.

```
set var 5
=> 5
set b $var
=> 5
```

The second `set` command above assigns to variable `b` the value of variable `var`. The use of the dollar sign is our first example of substitution. You can imagine that the second `set` command gets rewritten by substituting the value of `var` for `$var` to obtain a new command.

```
set b 5
```

The actual implementation is a little different, but not much.

Command Substitution

The second form of substitution is *command substitution*. A nested command is delimited by square brackets, `[` and `]`. The Tcl interpreter takes everything between the brackets and evaluates it as a command. It rewrites the outer command by replacing the square brackets and everything between them with the result of the nested command. This is similar to the use of backquotes in other shells, except that it has the additional advantage of supporting arbitrary nesting of other commands.

Example 1–3 Command substitution.

```
set len [string length foobar]
=> 6
```

In the example, the nested command is:

```
string length foobar
```

The `string` command performs various operations on strings Here we are asking for the length of the string `foobar`.

Command substitution causes the outer command to be rewritten as if it were:

```
set len 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered the command it delimits is evaluated.

Note that the spaces in the nested command are ignored for the purposes of grouping the arguments to `set`. In addition, if the result of the nested command contains any spaces or other special characters, they are not interpreted. These issues will be illustrated in more detail later in this chapter. The basic rule of thumb is that the interpreter treats everything from the left bracket to the matching right bracket as one lump of characters, and it replaces that lump with the result of the nested command.

Math Expressions

The `expr` command is used to evaluate math expressions. The Tcl interpreter itself has no particular smarts about math expressions. It treats `expr` just like any other command, and it leaves the expression parsing up to the `expr` implementation. The math syntax supported by `expr` is much like the C expression syntax, and a more complete summary of the expression syntax is given in the reference section at the end of this chapter.

The `expr` command primarily deals with integer, floating point, and boolean values. Logical operations return either 0 (false) or 1 (true). Integer values are promoted to floating point values as needed. Scientific notation for floating point numbers is supported. There is some support for string comparisons by `expr`, but the `string compare` command described in Chapter 2 is more reliable because `expr` may do conversions on strings that look like numbers.

Example 1–4 Simple arithmetic.

```
expr 7.2 / 3
=> 2.4
```

The implementation of `expr` takes all its arguments, concatenates them back into a single string, and then parses the string as a math expression. After `expr` computes the answer, the answer is formatted into a string and returned.

Example 1–5 Nested commands.

```
set len [expr [string length foobar] + 7]
=> 13
```

You can include variable references and nested commands in math expressions. The example uses `expr` to add `7` to the length of the string `foobar`. As a result of the inner-most command substitution, the `expr` command sees `6 + 7`, and `len` gets the value `13`.

Example 1–6 Built-in math functions.

```
set pi [expr 2*asin(1.0)]
=> 3.14159
```

The expression evaluator supports a number of built-in math functions. A complete listing is given on page 15. The example computes the value of *pi*.

By default, 6 significant digits are used when returning a floating point value. This can be changed by setting the `tcl_precision` variable to the number of significant digits desired. 17 digits of precision is enough to ensure that no information is lost when converting back and forth between a string and an IEEE double precision number.

Example 1–7 Controlling precision with tcl_precision.

```
expr 1 / 3
=> 0
expr 1 / 3.0
=> 0.333333
set tcl_precision 17
=> 17
expr 1 / 3.0
=> 0.33333333333333331
```

Backslash Substitution

The final type of substitution done by the Tcl interpreter is *backslash substitution*. This is used to quote characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. You can also specify characters that are hard to type directly by giving their octal or hexadecimal value.

As a rule, however, if you find yourself using lots of backslashes, there is probably a simpler way to achieve the effect you are striving for. For starters, you can group things with curly braces to turn off all interpretation of special characters. However, there are cases where backslashes are required.

Example 1–8 Quoting special characters with backslash.

```
set dollar \$
=> $
set x $dollar
=> $
```

In the example, the value of `dollar` does not affect the substitution done in the assignment to `x`. After the example, the value of `x` and `dollar` is the single character, `$`. This is a crucial property of the Tcl interpreter*: only a single round of interpretation is done*. You don't have to worry about variables with funny values.

You can also specify characters with their hex or octal value:

```
set escape \0x1b
set escape \033
```

The value of variable `escape` is the ASCII ESC character, which has character code `27`. The table on page 14 summarizes backslash substitutions.

Another common use of backslashes is to continue long commands on multiple lines. A backslash as the last character in a line is converted into a space. In addition, all the white space at the beginning of the next line is also absorbed by this substitution. Often line continuations can be avoided by strategic placement of opening curly braces as will be shown in the `proc` example below. However, the case where this does not work is with nested commands delimited by square brackets. Inside square brackets, the rule that newline and semi-colon are command terminators still applies. The backslash in the next example is required, otherwise the `expr` command would get terminated too soon, and the value of `[string length $two]` would be used as the name of a command!*

Example 1–9 Continuing long lines with backslashes.

```
set totalLength [expr [string length $one] + \
```

* The reasoning for this feature of the parse is consistency. A newline terminates a command unless an argument is being grouped. This holds for both top level and nested commands. The square brackets used for command substitution do not provide grouping. This allows the nested commands to be embedded as part of an argument.

```
        [string length $two]]
```

Double Quotes

Double quotes, like braces, are used to group words together. The difference between double quotes and curly braces is that quotes allow substitutions to occur in the group, while curly braces prevent substitutions.

Example 1–10 Grouping with double quotes allows substitutions.

```
set s Hello
puts stdout "The length of $s is [string length $s]."
=> The length of Hello is 5.
puts stdout {The length of $s is [string length $s].}
=> The length of $s is [string length $s].
```

In the first command of the example, the Tcl interpreter does variable and command substitution on the second argument to `puts`. In the second command, substitutions are prevented so the string is printed as is.

In practice, grouping with curly braces is used when substitutions on the argument need to be delayed until a later time (or never done at all). Examples include control flow statements and procedure declarations. Double quotes are useful in simple cases like the `puts` command above.

Another common use of quotes is with the `format` command that is similar to the C printf function. The first argument to `format` is a format specifier that often includes special characters like newlines, tabs, and spaces. The only way to effectively group these into a single argument to `format` is with quotes. The quotes allow the Tcl interpreter to do the backslash substitutions of \n and \t while ignoring spaces.

```
puts [format "Item: %s\t%5.3f" $name $value]
```

Here `format` is used to align a name and a value with a tab. The `%s` and `%5.3f` indicate how the remaining arguments to `format` are to be formatted. Note that the trailing `\n` usually found in a C `printf` call is not needed because `puts` provides one for us. More details about the `format` command can be found in Chapter 2.

Procedures

Tcl uses the `proc` command to define procedures. The basic syntax to define a procedure is:

```
proc name arglist body
```

The first argument is the name of the procedure being defined. The name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each other. The second argument is a list of parameters to the procedure. The third argument is a command, or more typically a group of commands that form the procedure body. Once defined, a Tcl procedure is used just like any of the built-in commands.

Example 1–11 Defining a procedure.

```
proc diag {a b} {
    set c [expr sqrt($a * $a + $b * $b)]
    return $c
}
```

The `diag` procedure defined in the example computes the length of the diagonal side of a right triangle given the lengths of the other two sides. The `sqrt` function is one of many math functions supported by the `expr` command. The variable `c` is local to the procedure; it is only defined during execution of `diag`. Variable scope is discussed further in Chapter 5. Use of this variable is not really necessary in this example. The procedure body could also be written as:

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

The `return` command is optional in this example because the Tcl interpreter will return the value of the last command in the body as the value of the procedure. So, the procedure body could be reduced to:

```
expr sqrt($a * $a + $b * $b)
```

Note the stylized use of curly braces in this example. Braces group the arguments `a` and `b` into a single argument list to form the second argument to the `proc` command. The curly brace at the end of the first line starts the third argument. In this case, the Tcl interpreter sees the opening left brace, causing it to ignore newline characters and gobble up text until a matching right brace is found. (Double quotes have the same property. They group characters, including newlines, until another double quote is found.) The result of the grouping is that the third argument to `proc` is a sequence of commands. When they are evaluated later, the embedded newlines will terminate each command. The other crucial effect of the curly braces around the procedure body is to delay any substitutions in the body until the time the procedure is called. For example, the variables `a`, `b` and `c` are not defined until the procedure is called, so we do not want to do variable substitution at the time `diag` is defined.

The `proc` command supports additional features such as having variable numbers of arguments and default values for arguments. These are described in detail in Chapter PROCS.

A While Loop Example

Let's reinforce what we've learned so far with a longer example.

Example 1–12 A loop that multiplies the numbers from 1 to 10.

```
set i 1 ; set product 1
while {$i <= 10} {
    set product [expr $product * $i]
    incr i
}
set product
=> 3628800
```

The semi-colon is used on the first line to remind you that it is a command terminator just like the newline character.

The example uses the `while` command to compute the product of a series of numbers. The first argument to `while` is a boolean expression, and its second argument is a sequence of commands, or *command body*, to execute. The `while` command will evaluate the boolean expression, and then execute the body if the expression is true (non-zero). The `while` command will continue to test the expression and then evaluate the command body until the expression is false (zero).

The same math expression evaluator used by the `expr` command is used by `while` to evaluate the boolean expression. There is no need to explicitly use the `expr` command in the first argument, even if you have a much more complex expression.

The `incr` command is used to increment the value of the loop variable `i`. The `incr` command can take an additional argument, a positive or negative integer by which to change the value of the variable. This is a handy command that saves us from the longer command:

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

Curly braces are used to group the two arguments to `while`. The loop body is grouped just like we grouped the procedure body earlier. The use of braces around the boolean expression is also crucial because it delays variable substitution until the `while` command implementation tests the expression. The following example is an infinite loop:

```
set i 1 ; while $i<=10 {incr i}
```

The loop will run indefinitely. The bug is that the Tcl interpreter will substitute for `$i` *before* `while` is called, so `while` gets a constant expression `1<=10` that will always be true. You can avoid these kinds of errors by adopting a consistent coding style that always groups expressions and command bodies with curly braces.

Expressions can include variable and command substitutions and still be grouped with curly braces because the expression parser does its own round of substitutions.* This is needed in the example if it is to obtain the current value of

* This means that an argument to expr can be subject to two rounds of substitution: one by the Tcl interpreter before expr is called, and a second by the implementation of expr itself. Ordinarily this is not a problem because math values do not contain the characters that are special to the Tcl interpreter. The fact that `expr` does substitutions on its argument internally means that it is OK to group its argument with curly braces.

`$i` in the boolean expression.

The last command in the example uses `set` with a single argument. When used in this way the `set` command returns the current value of the named variable.

Grouping And Command Substitution

The following example demonstrates how nested commands interact with grouping arguments to the main command. A nested command is treated as one lump of characters, regardless of its internal structure, so a nested command is always included with the surrounding group of characters when collecting arguments for the main command.

Example 1–13 Embedded command and variable substitution.

```
set x 7 ; set y 9
puts stdout $x+$y=[expr $x + $y]
=> 7+9=16
```

In the example the second argument to `puts` is:

```
$x+$y=[expr $x + $y]
```

The white space inside the nested command is ignored for the purposes of grouping the argument. The Tcl interpreter makes a single pass through the argument doing variable and command substitution. By the time it encounters the left bracket, it has already done some variable substitutions to obtain:

```
7+9=
```

At that point it calls itself recursively to evaluate the nested command. Again, the `$x` and `$y` are substituted before calling `expr`. Finally, the result of `expr` is substituted for everything from the left bracket to the right bracket. The `puts` command gets the following as its second argument:

```
7+9=16
```

The main point is that the grouping decision about `puts`'s second argument is made before the command substitution is done. Even if the result of the nested command contained spaces or other special characters, they would be ignored for the purposes of grouping the arguments to the outer command. If you wanted the output to look nicer, with spaces around the `+` and `=`, then you would use double quotes to explicitly group the argument to `puts`:

```
puts stdout "$x + $y = [expr $x + $y]"
```

In contrast, it is never necessary to explicitly group a nested command with double quotes if it makes up the whole argument. The following is a redundant use of double quotes:

```
puts stdout "[expr $x + $y]"
```

In general, you can place a bracketed command anywhere. The following computes a command name:

```
[findCommand $x] arg arg
```

The following concatenates the results of two commands because there is no whitespace between the `]` and `[`.

```
set x [cmd1 arg][cmd2 arg]
```

More About Variable Substitution

Grouping and variable substitution interact in the same way that grouping and command substitution do. Spaces or special characters in variable values do not affect grouping decisions because these decisions are made before the variable values are substituted. The rule of thumb is *grouping before substitution*.

Example 1–14 Embedded variable references

```
set foo filename
set object $foo.o
=> filename.o
set a AAA
set b abc${a}def
=> abcAAAdef
set .o yuk!
set x ${.o}y
=> yuk!y
```

The Tcl interpreter makes some assumptions about variable names that make it easy to embed their values into other strings. By default, it assumes that variable names only contain letters, digits, and the underscore. The construct `$foo.o` represents a concatenation of the value of `foo` and the literal "`.o`".

If the variable reference is not delimited by punctuation or whitespace, then you can use curly braces to explicitly delimit the variable name. This construct can also be used to reference variables with funny characters in their name (although you probably do not want variables named like that).

Substitution And Grouping Summary

The following rules summarize the fundamental mechanisms of grouping and substitution that are performed by the Tcl interpreter before it invokes a command:

1. A dollar sign, `$`, causes variable substitution. Variables names can be any length, and case is significant. If variable references are embedded into other strings, they can be distinguished with `${varname}` syntax.
2. Square brackets, `[ ]`, cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.

3. The backslash character, \, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character(s) are replaced with a new character.
4. Substitutions can occur anywhere (unless prevented by curly brace grouping). A substitution can occur in the middle of a word. That is, part of the word can be a constant string, and other parts of it can be the result of substitutions. Even the command name can be affected by substitutions.
5. Grouping with curly braces, `{ }`, prevents substitutions. Braces nest. The interpreter includes all characters between the matching left and right brace in the group, including newlines, semi-colons, and nested braces. The enclosing (i.e., outer-most) braces are not included in the group.
6. Grouping with double-quotes, `" "`, allows substitutions. The interpreter groups everything until another double-quote is found, including newlines and semi-colons. The enclosing quotes are not included in the group of characters. A double-quote character can be included in the group by quoting it with a backslash.
7. Grouping decisions are made before substitutions are performed. This means that the values of variables or command results do not affect grouping.
8. A single round of substitutions is performed before command invocation. That is, the result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that contains special characters such as spaces, dollar-signs, square brackets or braces. Because only a single round of substitution is done, you don't have to worry about special characters in values causing extra substitutions.

Fine Points

Here are some additional tips.

1. A well-formed Tcl list has whitespace, a left curly brace, or a left square bracket before each left curly brace. After a right curly brace you can have either another right brace, a right square bracket, or whitespace. This is because white space is used as the separator, while the braces only provide grouping. One common error is to forget a space between the right curly brace that ends one argument and the left curly brace that begins the next one.
2. A double-quote is only interesting when it comes after white space. That is, the interpreter only uses it for grouping in this case. As with braces, white space, a right bracket, or a right curly brace are the only things allowed after the closing quote.
3. Spaces are *not* required around the square brackets used for command sub-

stitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the same group.

4. When grouping with braces or double quotes, newlines and semi-colons are ignored for the purposes of command termination. They get included in the group of characters just like all the others.
5. During command substitution, newlines and semi-colons *are* significant as command terminators. If you have a long command that is nested in square brackets, put a backslash before the newline if you want to continue the command on another line.

Comments

Tcl uses the `#` character for comments. Unlike many languages, the `#` must occur at the beginning of a command. (Much like `REM` in Basic.) An easy trick to append a comment to the end of a command is to proceed the `#` with a semicolon in order to terminate the previous command.

```
# Here are some parameters
set rate 7.0;# The interest rate
set months 60;# The loan term
```

One subtle effect to watch out for is that a backslash effectively continues a comment line onto the next line of the script. In addition, a semi-colon inside a comment is not significant. Only a newline terminates comments.

```
# Here is the start of a Tcl comment \
and some more of it ; still in the comment
```

Command Line Arguments

The T cl shells pass the command line arguments to the script as the value of the `argv` variable. `argv` is a list, so you use the `lindex` command described in Chapter 3 to extract items from the argument list.

```
set first [lindex $argv 0]
set second [lindex $argv 1]
```

Table 1–5 on page 18 gives the complete set of pre-defined variables. Y ou can also use the `info vars` command to find out what is defined.

```
info vars
=> tcl_interactive argv0 argv auto_path argc env
```

Reference

Backslash Sequences

Table 1–1 Backslash sequences.

\a	Bell. (0x7)
\b	Backspace. (0x8)
\f	Form feed. (0xc)
\n	Newline. (0xa)
\r	Carriage return. (0xd)
\t	Tab (0x9)
\v	Vertical tab. (0xb)
\<newline>	Replace newline and all leading whitespace on the following line with a single space.
\\	Backslash. ('\')
ooo	Octal specification of character code. 1, 2, or 3 digits.
\x*hh*	Hexadecimal specification of character code. 1 or 2 digits.
c	Replaced with literal *c* if *c* is not one of the cases listed above. In particular , \$, \", \{ and \[are used to obtain these characters.

Arithmetic Operators

Table 1–2 Arithmetic Operators from highest to lowest precedence.

- ~ !	Unary minus, bitwise NOT, logical NOT.
* / %	Multiply, divide, remainder.
+ -	Add, subtract.
<< >>	Left shift, right shift.
< > <= >=	Comparison: less, greater, less or equal, greater or equal.
== !=	Equal, not equal.
&	Bitwise AND.
^	Bitwise NOT.
|	Bitwise OR.
&&	Logical AND.
||	Logical OR.
x?*y*:*z*	If *x* then *y* else *z*.

Built-in Math Functions

Table 1–3 Built-in Math functions

`acos(x)`	Arc-cosine of *x*.
`asin(x)`	Arc-sine of *x*.
`atan(x)`	Arc-tangent of *x*.
`atan2(y,x)`	Rectangular (*x*,*y*) to polar(*r*,*th*). atan2 gives *th*
`ceil(x)`	Least integral value greater than or equal to *x*.
`cos(x)`	Cosine of *x*.
`cosh(x)`	Hyperbolic cosine of *x*.
`exp(x)`	Exponential, e^x
`floor(x)`	Greatest integral value less than or equal to *x*.
`fmod(x,y)`	Floating point remainder of *x*/*y*.
`hypot(x,y)`	Returns `sqrt(x*x + y*y)`. *r* part of polar coordinates.
`log(x)`	Natural log of *x*.
`log10(x)`	Log base 10 of *x*.
`pow(x,y)`	x to the y power , x^y
`sin(x)`	Sine of *x*.
`sinh(x)`	Hyperbolic sine of *x*.
`sqrt(x)`	Square root of *x*.
`tan(x)`	Tangent of *x*.
`tanh(x)`	Hyperbolic tangent of *x*.
`abs(x)`	Absolute value of *x*.
`double(x)`	Promote *x* to floating point.
`int(x)`	Truncate *x* to an integer .
`round(x)`	Round *x* to an integer .

Core Tcl Commands

The pages given in Table 1–4 are the primary reference for the command.

Table 1–4 Built-in Tcl Commands

Command	Pg.	Description
`append`	21	Append arguments to a variable's value. No spaces added.

Table 1–4 Built-in Tcl Commands

`array`	37	Query array state and search through elements.
`break`	46	Premature loop exit.
`catch`	46	Trap errors.
`cd`	74	Change working directory.
`close`	74	Close an open I/O stream.
`concat`	32	concatenate arguments with spaces between. Splices lists together.
`continue`	46	Continue with next loop iteration.
`error`	48	Raise an error.
`eof`	70	Check for end-of-file.
`eval`	59	concatenate arguments and then evaluate them as a command.
`exec`	65	Fork and execute a UNIX program.
`exit`	75	Terminate the process.
`expr`	4	Evaluate a math expression.
`file`	67	Query the file system.
`flush`	70	Flush output from an I/O stream's internal buffers.
`for`	46	Loop construct similar to C `for` statement.
`foreach`	44	Loop construct over a list of values.
`format`	21	Format a string similar to C `sprintf`.
`gets`	72	Read a line of input from an I/O stream.
`glob`	74	Expand a pattern to matching file names.
`global`	53	Declare global variables.
`history`	81	Command-line history control.
`if`	42	Conditional command. Allows `else` and `elseif` clauses.
`incr`	8	Increment a variable by an integer amount.
`info`	77	Query the state of the Tcl interpreter.
`join`	35	concatenate list elements with a given separator string.
`lappend`	32	Add elements to the end of a list.
`lindex`	33	Fetch an element of a list.
`linsert`	34	Insert elements into a list.
`list`	32	Create a list out of the arguments.

Table 1–4 Built-in Tcl Commands

`llength`	33	Return the number of elements in a list.
`lrange`	33	Return a range of list elements.
`lreplace`	34	Replace elements of a list
`lsearch`	34	Search for an element of a list that matches a pattern.
`lsort`	35	Sort a list.
`open`	70	Open a file or process pipeline for I/O.
`pid`	75	Return the process ID.
`proc`	51	Define a Tcl procedure.
`puts`	72	Output a string to an I/O stream.
`pwd`	74	Return the current working directory.
`read`	73	Read blocks of characters from an I/O stream.
`regexp`	26	Regular expression matching.
`regsub`	28	Substitutions based on regular expressions.
`rename`	52	Change the name of a Tcl command.
`return`	49	Return a value from a procedure.
`scan`	23	Similar to the C `sscanf` function.
`seek`	73	Set the seek offset of an I/O stream.
`set`	3	Assign a value to a variable.
`source`	1	Evaluate the Tcl commands in a file.
`split`	35	Chop a string up into list elements.
`string`	19	Operate on strings.
`switch`	43	Multi-way branch.
`tell`	73	Return the current seek offset of an I/O stream.
`time`	87	Measure the execution time of a command.
`trace`	39	Monitor variable assignments.
`unknown`	89	Unknown command handler.
`unset`	30	Undefine variables.
`uplevel`	57	Execute a command in a different scope.
`upvar`	55	Reference a variable in a different scope.
`while`	45	A loop construct.

Predefined Variables

Table 1–5 Variables defined by *tclsh*.

`argc`	The number of command line arguments
`argv`	A list of the command line arguments
`argv0`	The name of the script being executed. If being used interactively , argv0 is the name of the shell program.
`env`	An array of the environment variables. See page 38.
`tcl_interactive`	True (one) if the *tclsh* is prompting for commands.
`tcl_prompt1`	If defined, this is a command that outputs the prompt. .
`tcl_prompt2`	If defined, this is a command that outputs the prompt if the current command is not yet complete.
auto_path	The search path for script library directories. See page 90.
auto_index	A map from command name to a Tcl command that defines it.
auto_noload	If set, the library facility is disabled.
auto_noexec	If set, the auto execute facility is disabled.
`geometry`	(*wish* only). The value of the `-geometry` argument.

Note that the tcl_prompt1 variable is not a string to print. Its value is invoked as a command that prints the string. This lets you be arbitrarily fancy in how you generate prompts, but it makes the simple case harder. Try this:

```
set tcl_prompt1 {puts -nonewline "yes master> "}
```

CHAPTER 2

Strings and Pattern Matching

This chapter describes string manipulation and pattern matching. Tcl commands described: `string`, `append`, `format`, `regexp`, `regsub`, `glob`.

Strings are the basic data item in Tcl, so it should not be surprising that there are a large number of commands to manipulate strings. A closely related topic is pattern matching, in which string comparisons are made more powerful by matching a string against a pattern. Tcl supports two styles of pattern matching. Glob matching is a simple matching similar to that used in many shell languages. Regular expression matching is more complex and also more powerful.

The string Command

The general syntax of the Tcl `string` command is:

```
string operation stringvalue otherargs
```

That is, `string`'s first argument determines what it does, its second argument is a string, and there may be additional arguments depending on the operation.

Some of the string commands involve character indices into the string. These count from zero. The `end` keyword refers to the last character in a string.

```
string range abcd 1 end
=> bcd
```

The following table summarizes the `string` command. Most of these commands are closely related to the string functions in the standard C library.

Table 2–1 The `string` command

`string compare` *`str1 str2`*	Compare strings lexicographically. Returns 0 if equal, -1 if *`str1`* sorts before *`str2`*, else 1.
`string first` *`str1 str2`*	Return the index in *`str2`* of the first occurrence of *`str1`*, or -1 if *`str1`* is not found.
`string index` *`string index`*	Return the character at the specified *`index`*.
`string last` *`str1 str2`*	Return the index in *`str2`* of the last occurrence of *`str1`*, or -1 if *`str1`* is not found.
`string length` *`string`*	Return the number of characters in string.
`string match` *`pattern str`*	Return 1 if *`str`* matches the *`pattern`*, else 0. Glob-style matching is used. See page 24
`string range` *`str i j`*	Return the range of characters in *`str`* from *`i`* to *`j`*.
`string tolower` *`string`*	Return *`string`* in lower case.
`string toupper` *`string`*	Return *`string`* in upper case.
`string trim` *`string ?chars?`*	Trim the characters in *chars* from both ends of *`string`*. *`chars`* defaults to whitespace.
`string trimleft` *`string ?chars?`*	Trim the characters in *chars* from the beginning of *`string`*. *`chars`* defaults to whitespace.
`string trimright` *`string ?chars?`*	Trim the characters in *chars* from the end of *`string`*. *`chars`* defaults to whitespace.
`string wordend` *`str ix`*	Return the index in *`str`* of the character after the word containing the character at index *`ix`*.
`string wordstart` *`str ix`*	Return the index in *`str`* of the first character in the word containing the character at index *`ix`*.

Strings And Expresssions

Strings can be compared with `expr` using the comparison operators. However, there are a number of subtle issues that can cause problems. First, you must quote the string value so the expression parser can identify it as a string type. Then you must quote the expression with curly braces to preserve the double quotes from being stripped off by the main interpreter.

```
if {$x == "foo"}
```

The killer, however, is that in spite of the quotes the expression evaluator first converts things to numbers if possible, and then converts them back if it detects a case of string comparison. This can lead to unexpected conversions between strings that look like hex or octal numbers.

```
if {"0xa" == "10"} { puts stdout ack! }
=> ack!
```

As a result, the only bombproof way to compare strings is with the `string` compare command. This command also operates quite a bit faster because the unnecessary conversions are eliminated. Like the C library strcmp function, string compare returns 0 if the strings are equal, -1 if the first string is lexicographically less than the second, or 1 if the first string is greater than the second.

Example 2–1 Comparing strings.

```
if {[string compare $s1 $s2] == 0} {
    # strings are equal
}
```

The append Command

The append command takes a variable name as its first argument, and then it concatenates its remaining arguments onto the current value of the named variable. The variable will be created if it did not already exist.

```
set xyzzy z
append xyzzy a b c
=> zabc
```

The command provides an efficient way to add things to the end of a string. It works by exploiting the memory allocation scheme used internally by Tcl that allocates extra space to allow for string growth.

The format Command

The format command is similar to the C printf function. It formats a string according to a format specification:

format *spec value1 value2* ...

The spec argument includes literals and keywords. The literals are placed in the result as is, while each keyword indicates how to format the corresponding argument. The keywords are introduced with a percent (%) that is followed by zero or more modifiers and terminates with a conversion specifier. Example keywords include %f for floating point, %d for integer and %s for string format. Use %% to obtain a single percent character.

The following is a brief sketch of the keyword specification syntax. The complete details can be found in the on-line manual page about format. The most general keyword specification for each argument contains up to 6 parts: a position specifier, flags, field width, precision, word length, and conversion character.

The examples in this section use double quotes around the format specification. This is a habit because often the format contains white space, so grouping is required, as well as backslash substitutions like \t or \n, and the quotes allow substitution of these special characters.

The conversion characters are listed in the table below .

Table 2–2 Format conversions

`d`	Signed integer
`u`	Unsigned integer
`i`	Signed integer . The argument may be in hex (0x) or octal (0) format.
`o`	Unsigned octal.
`x` or `X`	Unsigned hexadecimal. ‘ `x`’ gives lower -case results.
`c`	Map from an integer to the character it represents in ASCII.
`s`	A string.
`f`	Floating point number in the format `a.b`
`e` or `E`	Floating point number in scientific notation, `a.bE+-c`
`g` or `G`	Floating point number in either `%f` or `%e` format, whichever is shorter .

A position specifier is *`i$`*, which means take the value from argument *`i`* as opposed to the normally corresponding argument. The position counts from 1. If you group the format specification with double-quotes, you will need to quote the `$` with a backslash.

```
set lang 2
format "%${lang}\$s" one un uno
=> un
```

The position is useful for picking a string from a set, such as this simple language-specific example. The position is also useful if the same value is repeated in the formatted string. If a position is specified for one format keyword, it must be used for all of them.

The flags in a format are used to specify padding and justification. The format flag characters are summarized in the table below .

Table 2–3 format flags

–	Left justify the field.
+	Always include a sign, either + or -.
space	Proceed a number with a space, unless the number has a leading sign. Useful for packing numbers close together .
`0`	Pad with zeros.
#	Leading 0 for octal. Leading 0x for hex. Always include a decimal point in floating point. Do not remove trailing zeros (%g).

```
format "%#x" 20
=> 0x14
format "%#08x" 10
=> 0x0000000a
```

After the flags you can specify a minimum field width value. The value is padded to this width if needed, normally with spaces, optionally with zeros if the 0 flag is used.

```
format "%-20s %3d" Label 2
=> Label2
```

You can compute a field width and pass it to `format` as one of the arguments by using `*` as the field width specifier. In this case the next argument is used as the field width instead of the value, and the argument after that is the value that gets formatted.

```
set maxl 8
format "%-*s = %s" $maxl Key Value
=> KeyValue
```

The precision comes next, and it is specified with a period and a number. For `%f` and `%e` it indicates how many digits come after the decimal point. For `%g` it indicates the total number of significant digits used. For `%d` and `%x` it indicates how many digits will be printed, padding with zeros if necessary.

```
format "%6.2f %6.2d" 1 1
=>   1.00 01
```

(The storage length part comes last, but it is rarely useful because Tcl maintains all floating point values in double-precision, and all integers as words.)

If you want to preserve enough precision in a floating point number so that scanning in the number later will result in the same thing, use `%17g`. (This magic number applies to double-precision IEEE format.)

The scan Command

The `scan` command is like the C `sscanf` procedure. It parses a string according to a format specification and assigns values to variables. It returns the number of successful conversions it made. The general form of the command is given below:

```
scan string format var ?var? ?var? ...
```

The format for `scan` is nearly the same as in the `format` command. There is no `%u` scan format. The `%c` scan format converts one character to its binary value. Unlike the C `sscanf %c`, it does not allow a field width.

The `scan` format includes a set notation. Use square brackets to delimit a set of characters. The set matches one or more characters that are copied into the variable. A dash is used to specify a range. The following scans a field of all lowercase letters.

```
scan abcABC {%[a-z]} result
```

```
=> 1
set result
=> abc
```

If the first character in the set is a right square bracket, then it is considered part of the set. If the first character in the set is `^`, then characters *not* in the set match. Again, put a right square bracket right after the ^ to include it in the set. Nothing special is required to include a left square bracket in the set. As in the example shown above, you'll want to protect the format with braces, or use backslashes, because square brackets are special to the Tcl parser.

String Matching

The `string match` command implements *glob*-style pattern matching that is modeled after the filename pattern matching done by various UNIX shells. There are just 3 constructs used in glob patterns: match any number of any characters (`*`), match any single character (`?`), or match one of a set of characters (`[abc]`).* Any other characters in a pattern are taken as literals that must match the input exactly. To match all strings that begin with `a`.

```
string match a* alpha
=> 1
```

To match all two-letter strings:

```
string match ?? XY
=> 1
```

To match all strings that begin with either `a` or `b`:

```
string match {[ab]*} cello
=> 0
```

Be careful! Square brackets are also special to the Tcl interpreter, so you'll need to wrap the pattern up in curly braces to prevent it from being interpreted as a nested command.

Another approach is to put the pattern into a variable:

```
set pat {[ab]*x}
string match $pat box
=> 1
```

The pattern specifies a range of characters with the syntax `[x-y]`. For example, `[a-z]` represents the set of all lower-case letters, and `[0-9]` represents all the digits. This range is applied to the ASCII collating sequence.

Finally, if you need to include a literal `*`, `?`, or bracket in your pattern, preface it with a backslash.

```
string match {*\?} what?
=> 1
```

* The `string match` function does not support alternation in a pattern, such as the `{a,b,c}` syntax of the C-shell. The `glob` command, however, does support this form.

In this case the pattern is quoted with curly braces because the Tcl interpreter is also doing backslash substitutions. Without the braces, you would have to do the following:

```
string match *\\? ?
=> 1
```

Regular Expressions

The most powerful way to express patterns is with regular expressions. It has a general pattern specification syntax, which includes the ability to extract substrings from the matching string. This proves quite useful in picking apart data.

A pattern is a sequence of a literal character, a matching character, a repetition clause, an alternation clause, or a subpattern grouped with parentheses. The following table summarizes the syntax of regular expressions:

Table 2–4 Regular Expression Syntax

`.`	Matches any character
`*`	Matches zero or more.
`+`	Matches one or more.
`?`	Matches zero or one.
`( )`	Groups a sub-pattern. The repetition and alternation operators apply to the whole proceeding sub-pattern.
`|`	Alternation.
`[ ]`	Delimit a set of characters. Ranges are specified as [*x-y*]. If the first character in the set is `^`, then there is a match if the remaining characters in the set are *not* present.
`^`	Anchor the pattern to the beginning of the string. Only when first.
`$`	Anchor the pattern to the end of the string. Only when last.

A number of examples of regular expressions are given below. Any pattern than contains brackets, dollar sign, or spaces must be handled specially when used in a Tcl command. Typically I use curly braces around patterns, although the examples below do not quote anything.

The general wild-card character is the period, "`.`". It matches any single character. The following pattern matches all two-character strings.

```
..
```

The matching character can be restricted to a set of characters with the `[`*xyz*`]` syntax. Any of the characters between the two brackets is allowed to match. For example, the following matches either `Hello` or `hello`:

```
[Hh]ello
```

The matching set can be specified as a range over the ASCII character set with the `[x-y]` syntax, which is the same as with the glob mechanism. However, there is also the ability to specify the complement of a set. That is, the matching character can be anything except what is in the set. This is achieved with the `[^xyz]` syntax. Ranges and complements can be combined. The following matches anything except the upper and lowercase letters:

```
[^a-zA-Z]
```

Repetition is specified with `*`, for zero-or -more, `+`, for one-or -more, and `?`, for zero-or -one. These operators apply to the previous thing, which is either a matching character, which could involve the set syntax, or a subpattern grouped with parentheses. The following matches a string that contains `b` followed by zero or more `a`'s:

```
ba*
```

While the following matches a string that has one or more sequences of `ab`:

```
(ab)+
```

The pattern that matches anything is:

```
.*
```

Alternation is specified with "`|`". Another way to match either `Hello` or `hello` would be with:

```
hello|Hello
```

In general, a pattern does not have to match the whole string. If you need more control than this, then you can anchor the pattern to the beginning of the string by starting the pattern with `^`, or to the end of the string by ending the pattern with `$`. You can force the pattern to match the whole string by using both. All strings that begin with spaces or tabs are matched with the following.

```
^( |\t)+
```

Finally, if a pattern can match several parts of a string, the matcher takes the match that occurs earliest in the input string. Then, if there is more than one match from that same point, the matcher takes the longest possible match. The rule of thumb is "first, then longest".

The regexp Command

The `regexp` command provides direct access to the regular expression matcher. Its syntax is:

```
regexp ?flags? pattern string ?match sub1 sub2...?
```

The return value is 1 if some part of the string matches the pattern, it is 0 otherwise.

The *flags* are optional and constrain the match as follows. If `-nocase` is specified, then upper case characters in *string* are treated as lower case during the match. If `-indices` is specified, then the match variables described below will each contain a pair of numbers that are the indices that delimit the match within *string*. Otherwise, the matching string itself is copied into the match variables. Finally, if your pattern begins with `-`, then you can use `--` to separate the flags from the pattern.

The *pattern* argument is a regular expression as described in the previous section. If this contains `$` or `[`, you have to be careful. The easiest thing to do is group your patterns with curly braces. However, if your pattern contains backslash sequences like `\n` or `\t` you will have to group with double quotes so the Tcl interpreter can do those substitutions. You will have to use `\[` and `\$` in your patterns in that case.

If *string* matches *pattern*, then the results of the match are stored into the variables named in the command. These match variable arguments are optional. If present, *match* is set to be the part of the string that matched the pattern. The remaining variables are set to be the substrings of *string* that matched the corresponding subpatterns in *pattern*. The correspondence is based on the order of left parentheses in the pattern to avoid ambiguities that can arise from nested subpatterns.

Example 2–2 Regular expression to parse the `DISPLAY` environment variable.

```
set env(DISPLAY) corvina:0.1
regexp {([^:]*):} $env(DISPLAY) match host
=> 1
set match
=> corvina:
set host
=> corvina
```

The example uses `regexp` to pick the hostname out of the `DISPLAY` environment variable, which has the form:

```
hostname:display
```

The pattern involves a complementary set, `[^:]`, to match anything except a colon. It uses repetition, `*`, to repeat that zero or more times. Then, it groups that part into a subexpression with parentheses. The literal colon ensures that the `DISPLAY` value matches the format we expect. The part of the string that matches the pattern will be stored into the `match` variable. The part that we really want is what matches the subpattern, and that will be stored into `host`. The whole pattern has been grouped with braces to avoid the special meaning of the square brackets to the Tcl interpreter. Without braces it would be:

```
regexp (\[^:]*): $env(DISPLAY) match host
```

This is quite a powerful statement, and it is efficient. If we only had the `string` command to work with, we would have had to resort to the following, which takes roughly twice as long to interpret.

```
set i [string first : $env(DISPLAY)]
if {$i >= 0} {
    set host [string range $env(DISPLAY) 0 [expr $i-1]]
}
```

Multiple subpatterns are allowed. We can improve our pattern so that it extracts the screen part of the DISPLAY as well as the host:

```
regexp {([^:]*):(.+)} $env(DISPLAY) match host screen
```

The regsub Command

The `regsub` command is used to do string substitution based on pattern matching. Its syntax is:

```
regsub ?switches? pattern string subspec varname
```

The `regsub` command returns the number of matches and replacements, or 0 if there was no match. `regsub` copies *`string`* to *`varname`*, replacing occurrences of *`pattern`* with the substitution specified by *`subspec`*.

The optional switches include `-all`, which means to replace all occurrences of the pattern. Otherwise only the first occurrence is replaced. The `-nocase` switch means that upper-case characters in the string are converted to lowercase before matching. The `--` switch is useful if your pattern begins with `-`.

The replacement pattern, *`subspec`*, can contain literal characters as well as the following special sequences.

`&` is replaced with the string that matched the pattern.

`\1` through `\9` are replaced with the strings that match the corresponding subpatterns in *`pattern`*. As with `regexp`, the correspondence is based on the order of left parentheses in the pattern specification.

The following is used to replace a user's home directory with a ~:

```
regsub ^$env(HOME)/ $pathname ~/ newpath
```

The following is used to construct a C compile command line given a filename. The `\.` is used to specify a match against period.

```
regsub {([^\.]*)\.c} file.c {cc -c & -o \1.o} ccCmd
```

The value assigned to `ccCmd` is:

```
cc -c file.c -o file.o.
```

With an input pattern of `file.c` and a pattern of `{([^\.]*)\.c}`, the subpattern matches everything up to the first period in the input, or just `file`. The replacement pattern, `{cc -c & -o \1.o}` references the subpattern match with `\1`, and the whole match with `&`.

CHAPTER 3

Tcl Data Structures

This chapter describes two higher level data structures used in Tcl: lists and arrays.

The basic data structure in Tcl is a string. In addition, there are two higher-level data structures, lists and arrays. Lists are implemented as strings. Their structure is defined by the syntax of the string. The syntax rules are the same as for commands, and in fact commands are just a particular instance of lists. Arrays are variables that have an index. The index is a string value, so you can think of arrays as maps from one string (the index) to another string (the value of the array element).

As a rule, lists are ok when they are short, or when you are building up a command to be evaluated later. Arrays are more convenient and efficient for larger collections of data.

More About Variables

Before we dive into lists and arrays, let's consider simple variables in a bit more detail. The `set` command is used to define variables of any type. In addition, the `set` command will return the value of a variable if it is only passed a single argument. It treats that argument as a variable name and returns the current value of the variable. The dollar-sign syntax used to get the value of a variable is really just a short-hand for using the `set` command in this way.

Example 3–1 Using set to return a variable value.

```
set var {the value of var}
=> the value of var
set name var
=> var
set name
=> var
set $name
=> the value of var
```

This is a somewhat tricky example. In the last command , `$name` gets substituted with `var`. Then the `set` command returns the value of `var`, which is `the value of var`. Another way to achieve a level of indirection like this is with nested `set` commands. The last `set` command above can be written as follows

```
set [set name]
=> the value of var
```

The unset command

You can delete a variable with the `unset` command:

```
unset varName varName2 ...
```

Any number of variable names can be passed to the `unset` command. However , `unset` will raise an error if a variable is not already defined.

You can delete an entire array , or just a single array element with `unset`. Using `unset` on an array is a convenient way to clear out a big data structure.

Using info to find out about variables

The existence of a variable can be tested with the `info exists` command. For example, because `incr` requires that a variable exists, you might have to test for the existence of the variable first.

Example 3–2 Using `info` to determine if a variable exists.

```
if ![info exists foobar] {
    set foobar 0
} else {
    incr foobar
}
```

In Chapter 5 there is an example on page 56 that implements a new version of `incr` that handles this case.

Tcl Lists

Unlike list data structures in other languages, Tcl lists are just strings with a special interpretation. By definition, a Tcl list has the same structure as a Tcl command. That is, a list is simply a string with list elements separated by white space. Braces or quotes can be used to group words with whitespace into a single list element. Because of the relationship between lists and commands, the list-related commands are used often when constructing Tcl commands.

The string representation of lists in Tcl has performance implications. The string representation must be reparsed on each list access, so watch out for large lists. If you find yourself maintaining large lists that must be frequently accessed, consider changing your code to use arrays instead.

There are several Tcl commands related to lists, and these are described briefly in Table 2-1. Their use will be described in more detail via some examples.

Table 3–1 List-related commands

`list` *`arg1 arg2`* ...	Creates a list out of all its arguments.
`lindex` *`list i`*	Returns the *i*'th element from *`list`*.
`llength` *`list`*	Returns the number of elements in *`list`*.
`lrange` *`list i j`*	Returns the *i*'th through *j*'th elements from *`list`*.
`lappend` *`listVar arg arg ...`*	Append a elements to the value of *`listVar`*.
`linsert` *`list index arg arg ..`*	Insert elements into *`list`* before the element at position *`index`*. Returns a new list.
`lreplace` *`list i j arg arg ...`*	Replace elements *i* through *j* of *`list`* with the *`args`*. Returns a new list.
`lsearch` *`mode list value`*	Return the index of the element in *`list`* that matches the *`value`* according to the mode, which is `-exact`, `-glob`, or `-regexp`. `-glob` is the default. Return -1 if not found.
`lsort` *`switches list`*	Sort elements of the list according to the switches: `-ascii, -integer, -real, -increasing, -decreasing, -command` *`command`*. Returns a new list.
`concat` *`arg arg arg`* ...	Join multiple lists together into one list.
`join` *`list joinString`*	Merge the elements of a list together by separating them with *`joinString`*.
`split` *`string splitChars`*	Split a string up into list elements, using (and discarding) the characters in *`splitChars`* as boundaries between list elements.

Constructing Lists: list, lappend, and concat

The `list` command constructs a list out of its arguments such that there is one list element for each argument. This is a important command, although it might not seem like it at first glance, because it ensures that the resulting list has the proper syntax. If any of the arguments contain special characters, the `list` command adds quoting to ensure they are parsed as a single element of the resulting list.

Example 3–3 Constructing a list with the `list` command.

```
set x {1 2}
=> 1 2
set x
=> 1 2
list $x \$ foo
=> {1 2} {$} foo
```

One thing that can be confusing at first is that the braces used to group the list value into one argument to the `set` command are not part of the list value. In the example, the interpreter strips off the outer braces that are used to group the second argument to `set`. However , the `list` command adds them back, which could lead you to believe that the braces are part of `x`'s value, but they are not.

The `lappend` command is used to append elements to the end of a list. It is efficient because it takes advantage of extra space allocated at the end of lists. Like `list`, `lappend` preserves the structure of its arguments. That is, it may add braces to group the values of its arguments so they retain their identity as list elements when they are appended onto the string representation of the list. The new elements added by `lappend` are peers of the existing list elements in the variable.

Example 3–4 Using `lappend` to add elements to a list.

```
lappend new 1 2
=> 1 2
lappend new 3 "4 5"
=> 1 2 3 {4 5}
set new
=> 1 2 3 {4 5}
```

The `lappend` command is unique among the list-related commands because its first argument is the name of a list-valued variable, while all the other commands take list values as arguments. You can call `lappend` with the name of an undefined variable and the variable will be created.

The `concat` command is useful for splicing together lists. It works by concatenating its arguments together, separating them with spaces. This joins multiple lists into one where the top-level list elements in each input list are also top-level list elements (i.e. peers) in the resulting list.

Example 3–5 Using concat to splice together lists.

```
concat 1 {2 3} {4 5 6}
=> 1 2 3 4 5 6
```

It turns out that double quotes behave much like the `concat` command. The following example compares the use of `list`, `concat`, and double quotes.

Example 3–6 Double quotes compared to the `list` command.

```
set x {1 2}
=> 1 2
set y "$x 3"
=> 1 2 3
set y [concat $x 3]
=> 1 2 3
set z [list $x 3]
=> {1 2} 3
```

The distinction between list and concat becomes important when Tcl commands are built dynamically. The basic rule is that `list` and `lappend` preserve list structure, while `concat` (or double-quotes) eliminate one level of list structure. The distinction can be subtle because there are examples where `list` and `concat` return the same results. Unfortunately, this can lead to data-dependent bugs. Throughout the examples of this book you will see the `list` command used to safely construct lists. This issue is discussed more in Chapter 6.

Getting List Elements: llength, lindex, and lrange

The `llength` command returns the number of elements in a list.

```
llength {a b {c d} "e f g" h}
=> 5
```

The `lindex` command returns a particular element of a list. It takes an index; list indices count from zero. The keyword *end* means the last element, and it can be used with `lindex`, `linsert`, `lrange`, and `lreplace`.

```
lindex {1 2 3} 0
=> 1
```

The `lrange` command returns a range of list elements. It takes a list and two indices as arguments.

```
lrange {1 2 3 {4 5}} 2 end
=> 3 {4 5}
```

Modifying Lists: linsert and lreplace

The `linsert` command inserts elements into a list value at a specified index. If the index is 0 or less, then the elements are added to the front. If the index is equal to or greater than the length of the list, then the elements are appended to the end. Otherwise, the elements are inserted before the element that is current as position index.

`lreplace` is used to replace a range of list elements with new elements. If you don' t specify any new elements, you effectively delete elements from a list.

Example 3–7 Modifying lists with `linsert` and `lreplace`.

```
linsert {1 2} 0 new stuff
=> new stuff 1 2
set x [list a {b c} e d]
=> a {b c} e d
lreplace $x 1 2 B C
=> a B C d
lreplace $x 0 0
=> {b c} e d
```

Searching Lists: lsearch

`lsearch` returns the index of a value in the list, or -1 if it is not present. `lsearch` supports pattern matching in its search. Glob-style pattern matching is the default, and this can be disabled with the `-exact` flag. The semantics of the pattern matching done with the `-glob` and `-regexp` options is described in Chapter 2. In the example below, the glob pattern `l*` matches the value `list`.

```
lsearch {here is a list} l*
=> 3
```

The `lreplace` command is often used with `lsearch` to determine if the list already contains the elements. The example below uses `lreplace` to delete elements by not specifying any replacement list elements.

Example 3–8 Deleting a list element by value.

```
proc ldelete { list value } {
    set ix [lsearch -exact $list $value]
    if {$ix >= 0} {
        return [lreplace $list $ix $ix]
    } else {
        return $list
    }
}
```

Sorting Lists: lsort

You can sort a list in a variety of ways with `lsort`. The three basic types of sorts are specified with the `-ascii`, `-integer`, or `-real` options. The `-increasing` or `-decreasing` option indicate the sorting order. The default option set is `-ascii -increasing`. The list is not sorted in place. Instead, a new list value is returned.

You can provide your own sorting function for special-purpose sorting needs. For example, suppose you have a list of person names, where each element is itself a list containing the person's first name, middle name (if any), and last name. The default sort will sort by everyone's first name. If you want to sort by their last name, however, you need to supply a sorting function.

Example 3–9 Sorting a list using a comparison function.

```
proc mycompare {a b} {
    set alast [lindex $a [expr [llength $a]-1]]
    set blast [lindex $b [expr [llength $b]-1]]
    set res [string compare $alast $blast]
    if {$res != 0} {
        return $res
    } else {
        return [string compare $a $b]
    }
}
set list {{Brent B. Welch} {John Ousterhout} {Miles Davis}}
=> {Brent B. Welch} {John Ousterhout} {Miles Davis}
lsort -command mycompare $list
=> {Miles Davis} {John Ousterhout} {Brent B. Welch}
```

The `mycompare` procedure extracts the last element from each of its arguments and compares those. If they are equal, then it just compares the whole of each argument.

The split And join Commands

The `split` command takes a string and turns it into a list by breaking it at specified characters. The `split` command provides a robust way to turn input lines into proper Tcl lists. Even if your data has space-separated words, you should be very careful when using list operators on arbitrary input data. Otherwise, stray double-quotes or curly braces in the input can result in invalid list structure and errors in your script.

Example 3–10 Use split to turn input data into Tcl lists.

```
set line {welch:*:3116:100:Brent Welch:/usr/welch:/bin/csh}
split $line :
=> welch * 3116 100 {Brent Welch} /usr/welch /bin/csh
set line {this is "not a tcl list}
```

```
lindex $line 1
=> is
lindex $line 2
=> unmatched open quote in list
lindex [split $line] 2
=> "not
```

The default split character is white space. If there are multiple separator characters in a row, these result in empty list elements - the separators are not collapsed. The following command splits on commas, periods, spaces and tabs:

```
set line "\tHello, world."
split $line \ ,.\t
=> {} Hello {} world {}
```

The `join` command is the inverse of split. It takes a list value and reformats it with specified characters separating the list elements. In doing so, it will remove any curly braces from the string representation of the list that are used to group the top-level elements. For example:

```
join {1 {2 3} {4 5 6}} :
=> 1:2 3:4 5 6
```

Arrays

The other primary data structure that Tcl has is arrays. An array is a variable with a string-valued index, so you can think of an array as a mapping from strings to strings. Internally an array is implemented with a hash table, so the cost of accessing each element is about the same. (It is affected a little by the length of the index.)

The index of an array is delimited by parentheses. The index can have any string value, and it can be the result of variable or command substitution. Array elements are defined with `set`:

```
set arr(index) value
```

The value of an array element is obtained with `$` substitution:

```
set foo $arr(index)
```

Example 3–11 Using arrays.

```
set arr(0) 1
for {set i 1} {$i <= 10} {incr i} {
    set arr($i) [expr $i * $arr([expr $i-1])]
}
```

This example sets `arr(x)` to the product of `1 * 2 * ... * x`. The initial assignment of `arr(0)` defines `arr` as an array variable. It is an error to use a variable as both an array and a normal variable. The following would be an error after the previous example:

```
set arr 3
=> can't set "arr": variable is array
```

If you have complex indices, use a comma to separate different parts of the index. A void putting a space after the comma. It is legal, but a space in an index value will cause problems because *parenthesis are not used as a grouping mechanism* . The space in the index needs to be quoted with a backslash, or the whole variable reference needs to be grouped:

```
set {arr(I'm asking for trouble)} {I told you so.}
```

Of course, if the array index is stored in a variable, then there is no problem with spaces in the variable's value. The following works fine:

```
set index {I'm asking for trouble}
set arr($index) {I told you so.}
```

The name of the array can be the result of a substitution. If the name of the array is stored in another variable, then you must use `set` as shown in the last command below to reference the array elements. If you are trying to pass an array by name to a procedure, see the example on page 56, which uses a different solution.

Example 3–12 What if the name of the array is in a variable.

```
set name TheArray
=> TheArray
set ${name}(xyz) {some value}
=> some value
set x $TheArray(xyz)
=> some value
set x ${name}(xyz)
=> TheArray(xyz)
set x [set name](xyz)
=> some value
```

The array Command

The `array` command returns information about array variables, and it can be used to iterate through array elements.

Table 3–2 The array command

`array exists` *arr*	Returns 1 if *arr* is an array variable.
array get *arr*	Returns a list that alternates between an index and the corresponding array value.
array names *arr* ?*pattern*?	Return the list of all indices defined for *arr*, or those that match the string match *pattern*.
array set *arr list*	Initialize the array *arr* from *list*, which should have the same form as the list returned by `get`.

Table 3–2 The array command

`array size` *arr*	Return the number of indices defined for *arr*.
`array startsearch` *arr*	Return a search id key for a search through *arr*.
`array nextelement` *arr id*	Return the value of the next element in *array* in the search identified by *id*. Returns an empty string if no more elements remain in the search.
`array anymore` *arr id*	Returns 1 if more elements remain in the search.
`array donesearch` *arr id*	End the search identified by *id*.

The `array names` command is perhaps the most useful because it allows easy iteration through an array with a `foreach` loop. (`foreach` is described in more detail on page 44.)

```
foreach index [array names arr] { command body }
```

The order of the names returned by `array names` is arbitrary. It is essentially determined by the hash table implementation of the array. You can limit what names are returned by specifying a pattern argument. The pattern is the kind supported by the `string match` command, which is described on page 24.

It is also possible to iterate through the elements of an array one at a time using the search-related commands. The ordering is also random, and in practice I find the `foreach` over the results of `array names` much more convenient. If your array has an extremely large number of elements, or if you need to manage an iteration over long period of time, then the array search operations might be more appropriate.

The `array get` and `array set` operations are used to convert between an array and a list. The list returned by `array get` has an even number of elements. The first element is an index, and the next is the corresponding array value. The ordering of the indexes is arbitrary. The list argument to `array set` must have the same structure.

Example 3–13 Converting from an array to a list.

```
set fruit(best) kiwi
set fruit(worst) peach
set fruit(ok) banana
array get fruit
=> ok banana best kiwi worst peach
```

Environment Variables

In a UNIX environment, the processes environment variables are available through the global array `env`. The name of the environment variable is the index, e.g., `env(PATH)`, and the array element contains the current value of the environment variable. If assignments are made to `env`, then they result in changes to the corresponding environment variable.[*]

Example 3–14 `printenv` prints the environment variable values.

```
proc printenv { args } {
    global env
    set maxl 0
    if {[llength $args] == 0} {
        set args [lsort [array names env]]
    }
    foreach x $args {
        if {[string length $x] > $maxl} {
            set maxl [string length $x]
        }
    }
    incr maxl 2
    foreach x $args {
        puts stdout [format "%*s = %s" $maxl $x $env($x)]
    }
}
printenv USER SHELL TERM
=>
USER   = welch
SHELL  = /bin/csh
TERM   = tx
```

Tracing Variable Values

The `trace` command lets you register a command to be called whenever a variable is accessed, modified, or unset. This form of the command is:

`trace variable` *`name ops command`*

The *`name`* is a Tcl variable name, which can be a simple variable, an array, or an array element. If a whole array is traced, then the trace is invoked when any element is used according to *`ops`*. The *`ops`* argument is one or more of the letters `r`, for read traces, `w`, for write traces, and `u`, for unset traces. The *`command`* is executed when one of these events occurs. It is invoked as:

`command name1 name2 op`

The *`name1`* argument is the variable or array name. The *`name2`* argument is the name of the array index, or null if the trace is on a simple variable. If there is an unset trace on an entire array and the array is unset, then *`name2`* is also null. The value of the variable is not passed to the procedure. The `upvar`, `uplevel`, or `global` commands have to be used to make the variable visible in the scope of the trace command. These commands are described in more detail in Chapter 5.

The next example uses traces to implement a read-only variable. The value

[*] Environment variables are a collection of string-valued variables associated each a UNIX process. Environment variables are inherited by child processes, so programs run with the Tcl `exec` call will inherit the environment of the Tcl script.

is modified before the trace procedure is called, so another variable (or some other mechanism) is needed to preserve the original value.

Example 3–15 Tracing variables.

```
set x-orig $x
trace variable x wu FixupX
proc FixupX { varName index op } {
    upvar $varName var
    global x-orig
    switch $op {
        w {set var $x-orig}
        u {unset x-orig}
    }
}
```

This example merely overrides the new value with the saved valued. Another alternative is to raise an error with the `error` command. This will cause the command that modified the variable to return the error . Another common use of trace is to update a user interface widget in response to a variable change. Several of the Tk widgets have this feature built into them.

If more than one trace is set on a variable, then they are invoked in the reverse order; the most recent trace is executed first. If there is a trace on an array and on an array element, then the trace on the array is invoked first. The next example uses an array trace to dynamically create array elements.

Example 3–16 Creating array elements with array traces.

```
# make sure variable is an array
set dynamic() {}
trace variable dynamic r FixupDynamic
proc FixupDynamic {name index op} {
    global dynamic;# We know this is $name
    if ![info exists dynamic($index)] {
        set dynamic($index) 0
    }
}
```

Information about traces on a variable is returned with the `vinfo` option.

```
trace vinfo dynamic
=> {r FixDynamic}
```

A trace is deleted with the `vdelete` trace option, which has the same form as the `variable` option. For example, the trace in the previous example can be removed with the following command.

```
trace vdelete dynamic r FixupDynamic
```

CHAPTER 4

Control Flow Commands

This chapter describes the Tcl commands used for flow control: `if`, `switch`, `foreach`, `while`, `for`, `break` `continue`, `catch`, `error`, `return`.

Control flow in Tcl is achieved with commands, just like everything else. There are looping commands: `while`, `foreach`, and `for`. There are conditional commands: `if` and `switch`. There is an error handling command: `catch`. Finally, there are some commands to fine tune control flow: `break`, `continue`, `return`, and `error`.

A flow control command often has a command body that is executed later, either conditionally or in a loop. In this case, it is important to group the command body with curly braces to avoid substitutions at the time the control flow command is invoked. Group with braces, and let the control flow command trigger evaluation at the proper time. A flow control command returns the value of the last command it chose to execute.

Another pleasant property of curly braces is that they group things together while including newlines. The examples use braces in a way that is both readable and convenient for extending the flow control commands across multiple lines.

Commands like `if`, `for` and `while` involve boolean expressions. They use the `expr` command internally, so there is no need for you to invoke `expr` explicitly to evaluate their boolean test expressions.

If Then Else

The `if` command is the basic conditional command. If an expression is true then execute one command body, otherwise execute another command body. The second command body (the `else` clause) is optional. The syntax of the command is:

```
if boolean then body1 else body2
```

The `then` and `else` keywords are optional. In practice, I omit `then`, but use `else` as illustrated in the next example. I always use braces around the command bodies, even in the simplest cases.

Example 4–1 A conditional `if-then-else` command.

```
if {$x == 0} {
    puts stderr "Divide by zero!"
} else {
    set slope [expr $y/$x]
}
```

The style of this example takes advantage of the way the Tcl interpreter parses commands. Recall that newlines are command terminators, except when the interpreter is in the middle of a group defined by braces (or double quotes). The stylized placement of the opening curly brace at the end of the first and third line exploits this property to extend the `if` command over multiple lines.

The first argument to `if` is a boolean expression. As a matter of style this expression is grouped with curly braces. The expression evaluator will perform variable and command substitution on the expression for us. Using curly braces ensures that these substitutions are performed at the proper time. It is possible to be lax in this regard, with constructs like:

```
if $x break continue
```

This is a sloppy, albeit legitimate `if` command that will either break out of a loop or continue with the next iteration depending on the value of variable `x`. Instead, always use braces around the command bodies to avoid trouble later and to improve the readability of your code. The following is much better (use `then` if it suites your taste).

```
if {$x} { break } else { continue }
```

Chained conditionals can be created by using the `elseif` keyword.

Example 4–2 Chained conditional with `elseif`.

```
if {$key < 0} {
    incr range 1
} elseif {$key == 0} {
    return $range
} else {
    incr range -1
}
```

Any number of conditionals can be chained in this manner. However, the `switch` command provides a more powerful way to test multiple conditions.

Switch

The `switch` command is used to branch to one of many command bodies depending on the value of an expression. In addition, the choice can be made on the basis of pattern matching as well as simple comparisons. Pattern matching is discussed in more detail in Chapter 2. Any number of pattern-body pairs can be specified. If multiple patterns match, only the body of the first matching pattern is evaluated.

The general form of the command is:

`switch` *`flags value pat1 body1 pat2 body2`* `...`

You can also group all the pattern-body pairs into one argument:

`switch` *`flags value`* `{` *`pat1 body1 pat2 body2`* `... }`

There are four possible flags that determine how *`value`* is matched.

`-exact`	Match the *`value`* exactly to one of the patterns. (The default.)
`-glob`	Use glob-style pattern matching. See page 24.
`-regexp`	Use regular expression pattern matching. See page 25.
`--`	No flag (or end of flags). Useful when *`value`* can begin with `-`.

There are three approaches to grouping the pattern and body pairs. The differences among them have to do with the substitutions that are performed (or not) on the patterns. You will want to group the command bodies with curly braces so that substitution only occurs on the body with the pattern that matches the value.

The first style groups all the patterns and bodies into one argument. This makes it easy to group the whole command without worrying about newlines, and it suppresses any substitutions on the patterns.

Example 4–3 Using `switch` for an exact match.

```
switch -exact -- $value {
    foo { doFoo; incr count(foo) }
    bar { doBar; return $count(foo)}
    default { incr count(other) }
}
```

If the pattern associated with the last body is `default`, then this command body is executed if no other patterns match. Note that the `default` keyword only works on the last pattern-body pair. If you use the `default` pattern on an earlier body, it will be treated as a pattern to match the literal string `default`.

The second style is useful if you have variable references or backslash sequences in the patterns that you need to have substituted. However, you have to use backslashes to escape the newlines in the command.

Example 4–4 Using `switch` with substitutions in the patterns.

```
switch -regexp -- $value \
   ^$key { body1 }\
   \t### { body2 }\
   {[0-9]*} { body3 }
```

In this example the first and second patterns have substitutions performed to replace `$key` with its value and `\t` with a tab character . The third pattern is quoted with curly braces to prevent command substitution; square brackets are part of the regular expression syntax, too. (See page 25.)

A third style allows substitutions on the patterns without needing to quote newlines, but you will have to backslash any double-quotes that appear in the patterns or bodies.

Example 4–5 Using `switch` with all pattern body pairs grouped with quotes.

```
switch -glob -- $value "
   ${key}* { puts stdout \"Key is $value\" }
   X* -
   Y* { takeXorYaction $value }
"
```

If the body associated with a pattern is just “ -”, then the `switch` command “falls through” to the body associated with the next pattern. Any number of patterns can be tied together in this fashion.

Foreach

The `foreach` command loops over a command body assigning a loop variable to each of the values in a list. The syntax is:

```
foreach loopVar valueList commandBody
```

The first argument is the name of a variable, and the command body is executed once for each element in the loop with the loop variable taking on successive values in the list. The list can be entered explicitly , as in the next example:

Example 4–6 Looping with `foreach`.

```
set i 1
foreach value {1 3 5 7 11 13 17 19 23} {
   set i [expr $i*$value]
}
set i
=> 111546435
```

In the next example, a list-valued variable is used.

Example 4–7 Parsing command line arguments.

```
# argv is set by the Tcl shells
foreach arg $argv {
    switch -regexp -- $arg {
        -foo      {set fooOption 1}
        -bar      {barRelatedCommand}
        -([0-9]+) {scan -%d $arg intValue}
    }
}
```

The variable `argv` is set by the Tcl interpreter to be a list of the command line arguments given when the interpreter was started up. The loop looks for various command line options. The -- flag is *required* in this example because the `switch` command will complain about a bad flag if the pattern begins with a - character. The `scan` command, which is similar to the C library `scanf` function, is used to pick a number out of one argument.

If the list of values is to contain variable values or command results, then the `list` command should be used to form the list. Double-quotes should be avoided because if any values or command results contain spaces or braces, the list structure will be reparsed, which can lead to errors or unexpected results.

Example 4–8 Using `list` with `foreach`.

```
foreach x [list $a $b [foo]] {
    puts stdout "x = $x"
}
```

The loop variable `x` will take on the value of `a`, the value of `b`, and the result of the `foo` command, regardless of any special characters or whitespace in those values.

While

The `while` command takes two arguments, a test and a command body:

```
while booleanExpr body
```

The `while` command repeatedly tests the boolean expression and then executes the body if the expression is true (non-zero). Because the test expression is evaluated again before each iteration of the loop, it is crucial to protect the expression from any substitutions before the while command is invoked. The following is an infinite loop (See also Example 1-11 in Chapter 1):

```
set i 0 ; while $i<10 {incr i}
```

The following behaves as expected:

```
set i 0 ; while {$i<10} {incr i}
```

It is also possible to put nested commands in the boolean expression. The following example uses `gets` to read standard input. The `gets` command returns the number of characters read, returning -1 upon end-of-file. Each time through the loop the variable `line` contains the next line in the file.

Example 4–9 A `while` loop to read standard input.

```
set numLines 0 ; set numChars 0
while {[gets stdin line] >= 0} {
    incr numLines
    incr numChars [string length $line]
}
```

For

The `for` command is similar to the C `for` statement. It takes four arguments:

`for` *`initial test final body`*

The first argument is a command to initialize the loop. The second argument is a boolean expression that determines if the loop body will execute. The third argument is a command to execute after the loop body . Finally there is the loop body .

Example 4–10 A `for` loop.

```
for {set i 0} {$i < 10} {incr i 3} {
    lappend aList $i
}
set aList
=> 0 3 6 9
```

Break And Continue

Loop execution can be controlled with the `break` and `continue` commands. The `break` command causes immediate exit from a loop, while the `continue` command causes the loop to continue with the next iteration. Note that there is no `goto` statement in Tcl.

Catch

Until now we have ignored the possibility of errors. In practice, however, a command will raise an error if it is called with the wrong number of arguments, or if it detects some error condition particular to its implementation. If uncaught, an error will abort execution of a script.* The `catch` command is used to trap such errors. It takes two arguments:

```
catch command ?resultVar?
```

The first argument to `catch` is a command body. The second argument is the name of a variable that will contain the result of the command, or an error message if the command raises an error. `catch` returns 0 if there was no error caught, or 1 if it did catch an error.

It is important to use curly braces to group the command (as opposed to double-quotes) because `catch` will invoke the full Tcl interpreter on the command, so any needed substitutions will occur then. If double-quotes are used, an extra round of substitutions will occur before `catch` is even called.The simplest use of `catch` looks like the following.

```
catch { command }
```

A more careful `catch` phrase saves the result and prints an error message.

Example 4–11 A standard `catch` phrase.

```
if [catch { command arg1 arg2 ... } result] {
    puts stderr $result
} else {
    # command was ok, result is its return value
}
```

The most general `catch` phrase is shown in the next example. Multiple commands are grouped into a command body. The `errorInfo` variable is set by the Tcl interpreter after an error to reflect the stack trace from the point of the error.

Example 4–12 A longer `catch` phrase.

```
if [catch {
    command1
    command2
    command3
} result] {
    global errorInfo
    puts stderr $result
    puts stderr "*** Tcl TRACE ***"
    puts stderr $errorInfo
} else {
    # command body ok, result of last command is in result
}
```

These examples have not grouped the call to `catch` with curly braces. This is OK because `catch` always returns a 0 or a 1, so the `if` command will parse correctly. However, if we had used `while` instead of `if`, then curly braces would be necessary to ensure that the catch phrase was evaluated repeatedly.

* More precisely, the Tcl script will unwind and the current `Tcl_Eval` procedure will return `TCL_ERROR`. In Tk, errors that arise during event handling trigger a call to `tkerror`, a Tcl procedure you can implement in your application.

Error

The `error` command raises an error condition that will terminate a script unless it is trapped with the `catch` command. The command takes up to three arguments:

```
error message ?info? ?code?
```

The *message* becomes the error message stored in the result variable of the `catch` command.

If the *info* argument is provided, then the Tcl interpreter uses this to initialize the `errorInfo` global variable. That variable is used to collect a stack trace from the point of the error. If the *info* argument is not provided, then the `error` command itself is used to initialize the `errorInfo` trace.

Example 4–13 The results of `error` with no info argument.

```
proc foo {} {
    error bogus
}
foo
=> bogus
set errorInfo
=> bogus
    while executing
"error bogus"
    (procedure "foo" line 2)
    invoked from within
"foo"
```

In the example above, the `error` command itself appears in the trace. One common use of the `info` argument is to preserve the `errorInfo` that is available after a `catch`. The example below, the information from the original error is preserved.

Example 4–14 Preserving `errorInfo` when calling `error`.

```
if [catch {foo} result] {
    global errorInfo
    set savedInfo $errorInfo
    # Attempt to handle the error here, but cannot...
    error $result $savedInfo
}
```

The *code* argument is used to specify a concise, machine-readable description of the error. It gets stored into the global `errorCode` variable. It defaults to `NONE`. Many of the file system commands return an `errorCode` that contains starts with `POSIX` and contains the error code and associated message:

```
POSIX ENOENT {No such file or directory}
```

In addition, your application could define error codes of its own. Catch phrases could examine the code in the global errorCode variable and decide how to respond to the error.

Return

The return command is used to return from a procedure. It is needed if return is to occur before the end of the procedure body, or if a constant value needs to be returned. As a matter of style, I also use return at the end of a procedure, even though a procedure returns the value of the last command executed in the body.

Exceptional return conditions can be specified with some optional arguments to return. The complete syntax is:

```
return ?-code c? ?-errorinfo i? ?-errorcode ec? string
```

The -code option value is one of ok, error, return, break, continue, or an integer. ok is the default if -code is not specified.

The -code error option makes return behave much like the error command. In this case, the -errorcode option will set the global errorCode variable, and the -errorinfo option will initialize the errorInfo global variable.

Example 4–15 Specifying errorinfo with return.

```
proc bar {} {
    return -code error -errorinfo "I'm giving up" bogus
}
catch {bar} result
=> 1
set result
=> bogus
set errorInfo
=> I'm giving up
 invoked from within
"bar"
```

The return, break, and continue code options take effect in the caller of the procedure doing the exceptional return. If -code return is specified then the calling procedure returns. If -code break is specified, then the calling procedure breaks out of a loop, and if -code continue is specified then the calling procedure continues to the next iteration of the loop. Actually, with break and continue the interpreter will unwind the call stack until it finds a loop to operate on in these cases. These -code options to return are rarely used, although they enable the construction of new flow control commands entirely in Tcl.

CHAPTER 5

Procedures and Scope

Commands covered: `proc, global, upvar, uplevel`.

Procedures are used to parameterize a commonly used sequence of commands. In addition, each procedure has a new local *scope* for variables. The scope of a variable is the range of commands over which it is defined. This chapter describes the Tcl `proc` command in more detail, and then goes on to consider issues of variable scope.

The proc Command

A Tcl procedure is defined with the `proc` command. It takes three arguments:

```
proc name params body
```

The first argument is the procedure name, which will be added to the set of commands understood by the Tcl interpreter. The name is case sensitive, and can contain any characters at all. The second argument is a list of parameter names. The last argument is the body of the procedure.

Once defined, a Tcl procedure is used just like any other Tcl command. When it is called, each argument is assigned to the corresponding parameter and the body is evaluated. The result of the procedure is the result returned by the last command in the body. The `return` command can be used to return a specific value.

The parameter list for a procedure can include default values for parameters. This allows the caller to leave out some of the command arguments.

Example 5–1 Default parameter values.

```
proc p2 {a {b 7} {c -2} } {
    expr $a / $b + $c
}
p2 6 3
=> 0
```

Here the procedure `p2` can be called with one, two, or three arguments. If it is called with only one argument, then the parameters `b` and `c` will take on the values specified in the `proc` command. If two arguments are provided, then only `c` will get the default value, and the arguments will be assigned to `a` and `b`. At least one argument and no more than three arguments can be passed to `p2`.

A procedure can take a variable number of arguments by specifying the `args` keyword as the last parameter. When the procedure is called, the `args` parameter is a list that contains all the remaining values.

Example 5–2 Variable number of arguments.

```
proc argtest {a {b foo} args} {
    foreach param {a b args} {
        puts stdout "\t$param = [set $param]"
    }
}
argtest 1
=> a = 1
    b = foo
    args =
argtest 1 2
=> a = 1
    b = 2
    args =
argtest 1 2 3
=> a = 1
    b = 2
    args = 3
argtest 1 2 3 4
=> a = 1
    b = 2
    args = 3 4
```

Changing command names with rename

The `rename` command changes the name of a command.There are two main uses for `rename`. The first is to augment an existing procedure. Before you redefine it with `proc`, rename the existing command.

```
rename foo foo.orig
```

Then, from within the new implementation of `foo` you can invoke the original command as foo.orig. Existing users of foo will transparently use the new version.

The other thing you can do with rename is completely hide a command by renaming it to the empty string. For example, you might not want users to execute UNIX programs, so you could disable exec with the following command.

```
rename exec {}
```

Scope

There is a single, global scope for procedure names.[*]. You can define a procedure inside another procedure, but it is visible everywhere. There is a different name space for variables and procedures, so you could have a procedure and a variable with the same name without conflict.

Each procedure has a local scope for variables. That is, variables introduced in the procedure only live for the duration of the procedure call. After the procedure returns, those variables are undefined. Variables defined outside the procedure are not visible to a procedure, unless the upvar or global scope commands are used. If there is the same variable name in an outer scope, it is unaffected by the use of that variable name inside a procedure.

Example 5–3 Variable scope and Tcl procedures.

```
set a 5
set b -8
proc p1 {a} {
    set b 42
    if {$a < 0} {
        return $b
    } else {
        return $a
    }
}
p1 $b
=> 42
p1 [expr $a*2]
=> 10
```

There is no conflict between the variables a and b in the outer scope and either the parameter a or the local variable b.

The global Command

The top level scope is called the global scope. This scope is outside of any procedure. Variables defined at the global scope have to be made accessible to the com-

[*] This is in contrast to Pascal and other Algol-like languages that have nested procedures, and different than C that allows for file-private (static) procedures.

mands inside a procedure by using the `global` command. The syntax for global is:

```
global varName1 varName2 ...
```

Once a variable is made accessible with the `global` command, it is used just like any other variable. The variable does not have to be defined at the global scope when the `global` command is used. When the variable is defined, it will become visible in the global scope.

A useful trick is to collect your global variables into an array so that it is easier to manage your `global` statements. Even though you can put any number of variable names in the `global` command, it is tedious to update the various `global` commands when you introduce a new global variable. Using arrays, only a single `global` statement is needed. Another benefit of using arrays is that if you choose the array name to reflect the function of the collection of procedures that share the variables, (a *module* in other languages), then you will be less likely to have conflicts when you integrate your script with other code.

Example 5–4 A random number generator.*

```
proc randomInit { seed } {
 global rand
 set rand(ia) 9301  ;# Multiplier
 set rand(ic) 49297 ;# Constant
 set rand(im) 233280;# Divisor
 set rand(seed) $seed;# Last result
}
proc random {} {
 global rand
 set rand(seed) \
    [expr ($rand(seed)*$rand(ia) + $rand(ic)) % $rand(im)]
 return [expr $rand(seed)/double($rand(im))]
}
proc randomRange { range } {
   expr int([random]*$range)
}
randomInit [pid]
=> 5049
random
=> 0.517687
random
=> 0.217177
randomRange 100
=> 17
```

* Adapted from "Numerical Recipes in C" by Press et al. Cambridge University Press, 1988

Use Arrays for Global State

Tcl arrays are very flexible because there are no restrictions on the index value. A very good use for arrays is to collect together a set of related variables, much as one would use a record in other languages. An advantage of using arrays in this fashion is that a `global` scope command applies to the whole array, which simplifies the management of global variables.

For example, in a larger Tk application, each module of the implementation may require a few global state variables. By collecting these together in an array that has the same name as the module, name conflicts between different modules are avoided. Also, in each of the module's procedures, a single `global` statement will suffice to make all the state variables visible. More advanced scope control mechanisms are introduced by various object systems for Tcl, such as [incr tcl], which is described in Chapter 32.

The following artificial example uses an array to track the locations of some imaginary objects. (More interesting examples will be given in the context of some of the Tk widgets and applications.)

Example 5–5 Using arrays for global state.

```
proc ObjInit { o x y } {
    global obj
    set obj($o,x) $x
    set obj($o,y) $y
    set obj($o,dist) [expr sqrt($x * $x + $y * $y)]
}
proc ObjMove { o dx dy } {
    global obj
    if ![info exists obj($o,x)] {
        error "Object $o not initialized"
    }
    incr obj($o,x) $dx
    incr obj($o,y) $dy
    set obj($o,dist) [expr sqrt($obj($o,x) * $obj($o,x) + \
            $obj($o,y) * $obj($o,y))]
}
```

This example uses the global array `obj` to collect state variables, and it also parameterizes the index names with the name of an object. Remember to avoid spaces in the array indexes. The `incr` command and the `info exist` commands work equally well array elements as on scalar variables.

Call By Name Using upvar

The `upvar` command is used for situations in which you need to pass the name of a variable into a procedure as opposed to its value. Commonly this is used with array variables. The `upvar` command associates a local variable with a variable in a scope up the Tcl call stack. The syntax of the upvar command is:

```
upvar ?level? varName localvar
```

The *level* argument is optional, and it defaults to 1, which means one level up the Tcl call stack. You can specify some other number of frames to go up, or you can specify an absolute frame number with a `#number` syntax. Level `#0` is the global scope, so the `global foo` command is equivalent to:

```
upvar #0 foo foo
```

The variable in the uplevel stack frame can be either a scalar variable, an array element, or an array name. In the first two cases, the local variable is treated like a scalar variable. In the case of an array name, then the local variable is also treated like an array.

The following procedure uses `upvar` in order to print out the value of a scalar variable given its name. (See also Example 5–8 on page 56.)

Example 5–6 Print by name.

```
proc PrintByName { varName } {
    upvar $varName var
    puts stdout "$varName = $var"
}
```

`Upvar` can be used to fix `incr` procedure. One drawback of the built-in `incr` is that it raises an error if the variable does not exist. We can make a version of `incr` that will create the variable as needed.

Example 5–7 Improved `incr` procedure.

```
proc incr { varName {amount 1}} {
    upvar $varName var
    if [info exists var] {
        set var [expr $var + $amount]
    } else {
        set var $amount
    }
    return $var
}
```

Passing arrays by name

The `upvar` command words on arrays. You can pass an array name to a procedure and then use the `upvar` command to get an indirect reference to the array variable in the caller's scope. The next example illustrates this.

Example 5–8 Using an array to implement a stack.

```
proc Push { stack value } {
    upvar $stack S
```

```
        if ![info exists S(top)] {
            set S(top) 0
        }
        set S($S(top)) $value
        incr S(top)
    }
    proc Pop { stack } {
        upvar $stack S
        if ![info exists S(top)] {
            return {}
        }
        if {$S(top) == 0} {
            return {}
        } else {
            incr S(top) -1
            set x $S($S(top))
            unset S($S(top))
            return Sx
        }
    }
```

The array does not have to exist when the `upvar` command is called. The `Push` and `Pop` procedures both guard against a non-existent array with the `info exists` command. When the first assignment to `S(top)` is done by Push, the array variable is created in the caller's scope.

The uplevel Command

The `uplevel` command is similar to `eval`, except that it evaluates a command in a different scope than the current procedure. It is useful for defining new control structures entirely in Tcl. The syntax for `uplevel` is:

`uplevel` *`level command`*

As with `upvar`, the *`level`* parameter is optional and defaults to 1, which means to execute the command in the scope of the calling procedure. The other common use of level is `#0`, which means to evaluate the command in the global scope.

When you specify the *`command`* argument, you have to be aware of any substitutions that might be performed by the Tcl interpreter before `uplevel` is called. If you are entering the command directly, protect it with curly braces so that substitutions occur in the correct scope. The following affects the variable `x` in the caller's scope.

```
uplevel {set x [expr $x + 1]}
```

However, the following will use the value of `x` in the current scope to define the value of `x` in the calling scope, which is probably not what was intended:

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

It is also quite common to have the command in a variable. This is the case when the command has been passed into your new control flow procedure as an argument, or when you have built up the command using `list` and `lappend`. Or, perhaps you have read the command from a user-interface widget. In the control-flow case you most likely want to evaluate the command one level up:

```
uplevel $cmd
```

In the case of the user interface command, you probably want to evaluate the command at the global scope:

```
uplevel #0 $cmd
```

Finally, if you are assembling a command from a few different lists, such as the `args` parameter, then you'll have to use `concat` explicitly with `uplevel`:

```
uplevel [concat $cmd $args]
```

CHAPTER 6

Eval

This chapter describes explicit calls to the interpreter with the `eval` command. An extra round of substitutions is performed that results in some useful effects. The chapter describes the potential problems with `eval` and the ways to avoid them. The chapter also describes the `subst` command that does substitutions but no command invocation.

Evaluation involves substitutions, and it is sometimes necessary to go through an extra round of substitutions. This is achieved with the `eval` and `subst` commands. The need for more substitutions can crop up in simple cases, such as dealing with the list-valued `args` parameter to a procedure. In addition, there are commands like `after`, `uplevel`, and the Tk `send` command that have similar properties to `eval`, except that the command evaluation occurs later or in a different context.

The `eval` command is used to re-interpret a string as a command. It is very useful in certain cases, but it can be tricky to assemble a command so it is evaluated properly by `eval`. The root of the quoting problems is the internal use of `concat` by `eval` and similar commands to smash all their arguments into one command string. The result can be a loss of some important list structure so that arguments are not passed through as you expect. One general strategy to avoid these problems is to use `list` and `lappend` to explicitly form the command. In other cases, the `concat` is actually quite useful in joining together lists (e.g., `$args`) to make up a single command.

Eval And List

The `eval` command results in another call to the Tcl interpreter. If you construct a command dynamically, you will need to use `eval` to interpret it. For example, suppose we want to construct the following command now, but execute it later.

```
puts stdout "Hello, World!"
```

In this case, it is sufficient to do the following:

```
set cmd {puts stdout "Hello, World!"}
=> puts stdout "Hello, World!"
# sometime later...
eval $cmd
=> Hello, World!
```

However, suppose that the string to be output is stored in a variable, but that variable will not be defined at the time `eval` is used. We can artificially create this situation like this:

```
set string "Hello, World!"
set cmd {puts stdout $string}
unset string
eval $cmd
=> can't read "string": no such variable
```

The solution to this problem is to construct the command using `list`, as shown in the example below.

Example 6–1 Using `list` to construct commands.

```
set string "Hello, World!"
set cmd [list puts stdout $string]
=> puts stdout {Hello, World!}
unset string
eval $cmd
=> Hello, World!
```

The trick is that `list` has formed a list that has three elements: `puts`, `stdout`, and the value of `string`. The substitution of `$string` occurs before `list` is called, and `list` takes care of grouping that value for us.

In contrast, compare this to the most widely used incorrect approach:

```
set cmd "puts stdout $string"
=> puts stdout Hello, World!
eval $cmd
=> bad argument "World!": should be "nonewline"
```

The use of double quotes is equivalent to doing:

```
set cmd [concat puts stdout $string]
```

The problem here is that `concat` does not preserve list structure. The main lesson is that you should use `list` to construct commands if they contain variable values or command results that are substituted now as opposed to later on when the command is evaluated.

Eval And Concat

This section illustrates cases where `concat` is useful in assembling a command by concatenating multiple lists into one list. In fact, a `concat` is done internally by `eval` if it gets more than one argument.

```
eval list1 list2 list3 ...
```

The effect of `concat` is to join all the lists into one list; a new level of list structure is *not* added. This is useful if the lists are fragments of a command.

A common use for this form of `eval` is with the `args` construct in procedures. The `args` parameter can be used to layer functionality over another procedure. The new procedure takes optional arguments that are passed through to the lower layer. The problem with using `args`, however, is the proper formation of the call to the lower layer. The variable `args` has a bunch of arguments for the command, but they are all assembled into a list inside `args`.

This is illustrated with a simple Tk example. At this point, all you need to know is that a command to create a button looks like this:

```
button .foo -text Foo -command foo
```

After a button is created, it is made visible by packing it into the display:

```
pack .foo -side left
```

The following does not work:

```
set args {-text Foo -command foo}
button .foo $args
=> unknown option "-text Foo -command foo"
```

The problem is that `$args` is a list value, and `button` gets the whole list as a single argument. Instead, `button` needs to get the elements of `$args` as individual arguments. In this case, you can use `eval` and rely on the fact that it will concatenate its arguments and form a single list before evaluating things. The single list is, by definition, the same as a single Tcl command, so the `button` command parses correctly.

```
eval button .foo $args
=> .foo
```

Example 6–2 Using `eval` with `$args`.

```
# PackedButton creates and packes a button.
proc PackedButton {path txt cmd {pack {-side right}} args} {
    eval {button $path -text $txt -command $cmd} $args
    eval {pack $path} $pack
}
```

In `PackedButton`, both `pack` and `args` are list-valued parameters that are used as parts of a command. The internal `concat` done by `eval` is perfect for this situation. The simplest call to `PackedButton` is given below.

```
PackedButton .new "New" { New }
```

The quotes and curly braces are redundant in this case, but are retained to convey some type information. The pack argument takes on its default value, and the args variable is an empty list. The two commands executed by `PackedButton` are:

```
button .new -text New -command New
pack .new -side right
```

`PackedButton` creates a horizontal stack of buttons by default. The packing can be controlled with a packing specification:

```
PackedButton .save "Save" { Save $file } {-side left}
```

This changes the pack command to be:

```
pack .new -side left
```

The remaining arguments, if any, are passed through to the button command. This lets the caller fine tune some of the button attributes:

```
PackedButton .quit Quit { Exit } {-side left -padx 5} \
    -background red
```

This changes the button command to be:

```
button .new -text New -command New -background red
```

Double-quotes and eval

You may be tempted to use double-quotes instead of curly braces in your uses of `eval`. *Don't give in!* The use of double-quotes will probably be wrong. Suppose the first `eval` command were written like this:

```
eval "pack $path -text $txt -command $cmd $args"
```

This happens to work with the following because `txt` and `cmd` are one-word arguments with no special characters in them.

```
PackedButton .quit Quit { Exit }
```

In the next call an error is raised, however.

```
PackedButton .save "Save" { Save $file }
=> can't read "file": no such variable
```

The danger is that the success of this approach depends on the value of the parameters. The value of `txt` and the value of `cmd` are subject to another round of substitutions and parsing. When those values contain spaces or special characters, the command gets parsed incorrectly.

To repeat, the safe construct is:

```
eval {pack $path -text $txt -command $cmd} $args
```

As you may be able to tell, this was one of the more difficult lessons I learned, in spite of three uses of the word "concatenate" in the `eval` man page!

Commands That Concat Their Arguments

The `uplevel` command and two Tk commands, `after` and `send`, concatenate their arguments into a command and execute it later in a different context.

Whenever I discover such a command I put it on my danger list and make sure I explicitly form a single command argument with `list` instead of letting the command `concat` things together for me.

Get in the habit now:

```
after 100 [list doCmd $param1 $param2]
send $interp [list doCmd $param1 $param2];# Safe!
```

The worst part of this is that `concat` and `list` can result in the same thing, so you can be led down the rosy garden path, only to be bitten later when values change on you. The above two examples will always work. The next two will only work if `param1` and `param2` have values that are single list elements:

```
after 100 doCmd $param1 $param2
send $interp doCmd $param1 $param2;# Unsafe!
```

If you use other Tcl extensions that provide eval-like functionality, carefully check their documentation to see if they contain procedures that `concat` their arguments into a command. For example, Tcl-DP, which provides a network version of `send`, `dp_send`, also uses `concat`.

The subst Command

The `subst` command is used to do command and variable substitution, but without invoking any command. It is similar to `eval` in that it does a round of substitutions for you. However, it doesn't try to interpret the result as a command.

```
set a "foo bar"
subst {a=$a date=[exec date]}
=> a=foo bar date=Thu Dec 15 10:13:48 PST 1994
```

The `subst` command does not honor the quoting effect of curly braces. Instead, it will expand any variables or nested commands whereever they occur in its input argument.

```
subst {a=$a date={[exec date]}}
=> a=foo bar date={Thu Dec 15 10:15:31 PST 1994}
```

You can use backslashes to prevent variable and command substitution, though.

```
subst {a=\$a date=\[exec date]}
=> a=$a date=[exec date]
```

CHAPTER 7

Working with UNIX

This chapter describes how to use Tcl in a UNIX environment. Tcl commands: `exec`, `open`, `close`, `read`, `write`, `seek`, `tell`, `glob`, `pwd`, `cd`.

This chapter describes how to run programs and access the file system from Tcl. While these commands were designed for UNIX, they are also implemented (perhaps with limitations) in the Tcl ports to other systems such as DOS and Macintosh. These capabilities enable your Tcl script to be a general purpose glue that assembles other programs into a tool that is customized for your needs.

Running Unix Programs With exec

The `exec` command is used to run other UNIX programs from your Tcl script.* For example:

```
set d [exec date]
```

The standard output of the program is returned as the value of the `exec` command. However, if the program writes to its standard error stream or exits with a non-zero status code, then `exec` will raise an error.

The `exec` command supports a full set of *I/O redirection* and *pipeline* syntax. Each UNIX process normally has three I/O streams associated with it: standard input, standard output, and standard error. With I/O redirection you can

* Unlike the C-shell `exec` command, the Tcl exec does not replace the current process with the new one. Instead, the Tcl library forks first and executes the program as a child process.

divert these I/O streams to files or to I/O streams you have opened with the Tcl open command. A pipeline is a chain of UNIX processes that have the standard output of one command hooked up to the standard input of the next command in the pipeline. Any number of programs can be linked together into a pipeline.

Example 7–1 Using exec on a process pipeline.

```
set n [exec sort < /etc/passwd | uniq | wc -l 2> /dev/null]
```

The example uses exec to run three programs in a pipeline. The first program is sort, which takes its input from the file /etc/passwd. The output of sort is piped into uniq, which suppresses duplicate lines. The output of uniq is piped into wc, which counts up the lines for us. The error output of the command is diverted to the null device in order to suppress any error messages.

Table 7–1 gives a summary of the syntax understood by the exec command. Note that a trailing & causes the program to run in the background. In this case the process id is returned by the exec command. Otherwise, the exec command blocks during execution of the program and the standard output of the program is the return value of exec. The trailing newline in the output is trimmed off, unless you specify -keepnewline as the first argument to exec.

Table 7–1 Summary of the exec syntax for I/O redirection.

`-keepnewline`	(First arg only.) Do not discard trailing newline from the result.
`|`	Pipe standard output from one process into another.
`|&`	Pipe both standard output and standard error output.
`<` *fileName*	Take input from the named file.
`<@` *fileId*	Take input from the I/O stream identified by *fileId*.
`<<` *value*	Take input from the given value.
`>` *fileName*	Overwrite *fileName* with standard output.
`2>` *fileName*	Overwrite *fileName* with standard error output.
`>&` *fileName*	Overwrite *fileName* with both standard error and standard out.
`>>` *fileName*	Append standard output to the named file.
`2>>` *fileName*	Append standard error to the named file.
`>>&` *fileName*	Append both standard error and standard output to the named file.
`>@` *fileId*	Direct standard output to the I/O stream identified by *fileId*.
`2>@` *fileId*	Direct standard error to the I/O stream identified by *fileId*.
`>&@` *fileId*	Direct both standard error and standard output to the I/O stream.
`&`	As the last argument, indicates pipeline should run in background.

If you look closely at the I/O redirection syntax, you'll see that it is built up from a few basic building blocks. The basic idea is that '|' stands for pipeline, '>' for output, and '<' for input. The standard error is joined to the standard output by '&'. Standard error is diverted separately by using '2>'. You can use your own I/O streams by using '@'.

auto_noexec

The Tcl shell programs are set up by default to attempt to execute unknown Tcl commands as UNIX programs. For example, you can get a directory listing by typing:

```
ls
```

instead of

```
exec ls
```

This is handy if you are using the Tcl interpreter as a general shell. It can also cause unexpected behavior when you are just playing around. To turn this off, define the `auto_noexec` variable:

```
set auto_noexec anything
```

Looking At The File System

The Tcl `file` command provides several ways to check on the status of files in the UNIX file system. For example, you can find out if a file exists and what type of file it is. In fact, essentially all the information returned by the `stat` system call is available via the `file` command. Table 7–2 gives a summary of the various forms of the `file` command.

Table 7–2 The Tcl `file` command options.

`file atime` *name*	Return access time as a decimal string.
`file dirname` *name*	Return parent directory of file *name*.
`file executable` *name*	Return 1 if *name* has execute permission, else 0.
`file exists` *name*	Return 1 if *name* exists, else 0.
`file extension` *name*	Return the part of *name* from the last dot '.' to the end.
`file isdirectory` *name*	Return 1 if *name* is a directory, else 0.
`file isfile` *name*	Return 1 if *name* is not a directory, symbolic link, or device, else 0.
`file lstat` *name var*	Place stat results about the link *name* into *var*.
`file mtime` *name*	Return modify time of *name* as a decimal string.
`file owned` *name*	Return 1 if current user owns the file *name*, else 0.

Table 7–2 The Tcl `file` command options.

`file readable` *name*	Return 1 if *name* has read permission, else 0.
`file readlink` *name*	Return the contents of the symbolic link *name*.
`file rootname` *name*	Return all but the extension ('.' and onwards) of *name*.
`file size` *name*	Return the number of bytes in *name*.
`file stat` *name var*	Place stat results about *name* into array *var*. The elements defined for *var* are: `atime`, `ctime`, `dev`, `gid`, `ino`, `mode`, `mtime`, `nlink`, `size`, `type`, and `uid`.
`file tail` *name*	Return all characters after last '/' in *name*.
`file type` *name*	Return type identifier, which is one of: `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.
`file writable` *name*	Return 1 if *name* has write permission, else 0.

The following command uses `file mtime` to compare the modify times of two files. *

Example 7–2 A procedure to compare file modify times.

```
proc newer { file1 file2 } {
    expr [file mtime $file1] > [file mtime $file2]
}
```

A few of the options operate on pathnames as opposed to returning information about the file itself. You can use these commands on any string; there is no requirement that the pathnames refer to an existing file. The `dirname` and `tail` options are complementary. The first returns the parent directory of a pathname, while `tail` returns the trailing component of the pathname. For a simple pathname with a single component, the `dirname` option returns ".", which is the name of the current directory.

```
file dirname /a/b/c
=> /a/b
file tail /a/b/c
=> c
```

The `extension` and `root` options are also complementary. The `extension` option returns everything from the last period in the name to the end (i.e., the file suffix.) The `root` option returns everything up to, but not including, the last period in the pathname.

```
file root /a/b.c
```

* If you have ever resorted to piping the results of ls -l into awk in order to derive this information in other shell scripts, you'll appreciate these options.

```
=> /a/b
file extension /a/b.c
=> .c
```

The `makedir` example given below uses the `file` command to determine if it necessary to create the intermediate directories in a pathname. It calls itself recursively, using `file dirname` in the recursive step in order to create the parent directory. To do the actual work, it execs the `mkdir` program. An error can be raised in two places, explicitly by the makedir procedure if it finds a non-directory in the pathname, or by the mkdir program if, for example, the user does not have the permissions to create the directory.

Example 7–3 Creating a directory recusively.

```
proc makedir { pathname } {
	if {[file isdirectory $pathname]} {
		return $pathname
	} elseif {[file exists $pathname]} {
		error "Non-directory $pathname already exists."
	} else {
		# Recurse to create intermediate directories
		makedir [file dirname $pathname]
		exec mkdir $pathname
		return $pathname
	}
}
```

The most general `file` command options are `stat` and `lstat`. They take a third argument that is the name of an array variable, and they initialize that array with elements and values corresponding to the results of the `stat` system call. The array elements defined are: `atime, ctime, dev, gid, ino, mode, mtime, nlink, size, type`, and `uid`. All the element values are decimal strings, except for `type`, which can have the values returned by the `type` option. (See the `UNIX` man page on the `stat` system call for a description of these attributes.)

Example 7–4 Determining if pathnames reference the same file.

```
proc fileeq { path1 path2 } {
	file stat $path1 stat1
	file stat $path2 stat2
	expr [$stat1(ino) == $stat2(ino) && \
			$stat1(dev) == $stat2(dev)]
}
```

The example uses the device (`dev`) and inode (ino) attributes of a file to determine if two pathnames reference the same file.

Input/Output

The table below lists the commands associated with file input/output.

Table 7–3 Tcl commands used for file access.

`open` *what* ?*access*? ?*permissions*?	Open a file or pipeline.
`puts ?-nonewline?` ?*stream*? *string*	Write a string.
`gets` *stream* ?*varname*?	Read a line.
`read ?-nonewline?` *stream ?numBytes*?	Read bytes.
`tell` *stream*	Return the seek offset.
`seek` *stream offset ?origin*?	Set the seek offset. *origin* is one of `start`, `current`, or `end`.
`eof` *stream*	Query end-of-file status.
`flush` *stream*	Write out buffers of a stream.
`close` *stream*	Close an I/O stream.

Opening Files For I/O

The `open` command sets up an I/O stream to either a file or a pipeline of processes. The basic syntax is:

`open` *what ?access? ?permissions?*

The *what* argument is either a file name or a pipeline specification similar to that used by the `exec` command. The *access* argument can take two forms, either a short character sequence that is compatible with the `fopen` library routine, or a list of `POSIX` access flags. Table 7–4 summarizes the first form, while Table 7–5 summarizes the `POSIX` flags. If *access* is not specified, it defaults to read. The *permissions* argument is a value used for the permission bits on a newly created file. The default permission bits are `0666.` Consult the man page on the `UNIX chmod` command for more details about permission bits.

Example 7–5 Opening a file for writing.

```
set fileId [open /tmp/foo w 0600]
puts $fileId "Hello, foo!"
close $fileId
```

The return value of `open` is an identifier for the I/O stream. You use this in the same way the `stdout`, `stdin`, and `stderr` identifiers have been used in the examples so far, except that you need to store the result of `open` in a variable.

(You should consult your system's man page for the `open` system call to determine the precise effects of the `NOCTTY` and `NONBLOCK` flags.)

Table 7–4 Summary of the `open` access arguments.

r	Open for reading. The file must exist.
r+	Open for reading and writing. The file must exist.
w	Open for writing. Truncate if it exists. Create if it does not exist.
w+	Open for reading and writing. Truncate or create.
a	Open for writing. The file must exist. Data is appended to the file.
a+	Open for reading and writing. File must exist. Data is appended.

Table 7–5 Summary of `POSIX` flags for the access argument.

`RDONLY`	Open for reading.
`WRONLY`	Open for writing.
`RDWR`	Open for reading and writing.
`APPEND`	Open for append.
`CREAT`	Create the file if it does not exist.
`EXCL`	If `CREAT` is specified also, then the file cannot already exist.
`NOCTTY`	Prevent terminal devices from becoming the controlling terminal.
`NONBLOCK`	Do not block during the open.
`TRUNC`	Truncate the file if it exists.

Below is an example of how you'd use a list of `POSIX` access flags to open a file for reading and writing, creating it if needed, and not truncating it, which is something you cannot do with the simpler form of the access argument.

Example 7–6 Opening a file using the `POSIX` access flags.

```
set fileId [open /tmp/bar {RDWR CREAT}]
```

In general you want to be careful to check for errors when opening files. The following example illustrates a `catch` phrase used to open files. Recall that `catch` returns 1 if it catches an error, otherwise it returns zero. It treats its second argument as the name of a variable. In the error case it puts the error message into the variable. In the normal case it puts the result of the command into the variable.

Example 7–7 A more careful use of `open`.

```
if [catch {open /tmp/data r} fileId] {
   puts stderr "Cannot open /tmp/data: $fileId"
```

```
} else {
   # Read and process the file, then...
   close $fileId
}
```

Opening a process pipeline is done by specifying the pipe character, '|', as the first character of the first argument. The remainder of the pipeline specification is interpreted just as with the `exec` command, including input and output redirection. The second argument determines which end of the pipeline you get back from the open. The example below sorts the password file, and it uses the `split` command to separate the file lines into list elements.

Example 7–8 Opening a process pipeline.

```
set input [open "|sort /etc/passwd" r]
set contents [split [read $input] \n]
close $input
```

You can open a pipeline for both read and write by specifying the `r+` access mode. However, in this case you need to worry about buffering.After a `puts` the data may still be in a buffer in the Tcl library. Use the `flush` command to force this data out to the spawned processes before you try to read any output from the pipeline. In general, the `expect` extension, which is described in Chapter EXPECT, provides a much more powerful way to do these kinds of things.

Reading And Writing

The standard UNIX I/O streams are already open for you. These streams are identified by `stdin`, `sdout`, and `stderr`, respectively. Other I/O streams are identified by the return value of the `open` command. There are several commands used with file identifiers.

The puts and gets commands

The `puts` command writes a string and a newline to the output stream. There are a couple of details about the `puts` command that have not been used yet. It takes a `-nonewline` argument that prevents the newline character that is normally appended to the output stream. This will be used in the prompt example below. The second feature is that the stream identifier is optional, defaulting to `stdout` if not specified.

Example 7–9 Prompting for input.

```
puts -nonewline "Enter value: "
set answer [gets stdin]
```

The `gets` command reads a line of input, and it has two forms. In the example above, with just a single argument, `gets` returns the line read from the specified I/O stream. It discards the trailing newline from the return value. If end-of-file is reached, an empty string is returned. You have to use the `eof` command to tell the difference between a blank line and end-of-file. (`eof` returns 1 if there is end-of-file.) Given a second *`varName`* argument, `gets` stores the line into named variable and returns the number of bytes read. It discards the trailing newline, which is not counted. A -1 is returned if the stream has reached end of file.

Example 7–10 A read loop using `gets`.

```
while {[gets $stream line] >= 0} {
    # Process line
}
close $stream
```

The read command

The `read` command is used to read blocks of data, which can often be more efficient. It isn't clear in the table, but with `read` you can specify either the `-nonewline` argument or the *`numBytes`* argument, but not both. Without *`numBytes`*, the whole file (or what is left in the I/O stream) is read and returned. The `-nonewline` argument causes the trailing newline to be discarded. Given a byte count argument, read returns that amount, or less if not enough data remains in the stream. The trailing newline is not discarded in this case.

Example 7–11 A read loop using `read` and `split`.

```
foreach line [split [read $stream] \n] {
    # Process line
}
close $stream
```

For moderately sized files it is slightly faster, by about 10%, to loop over the lines in a file using the read loop in the second example. In this case, `read` is used to return the whole file, and `split` is used to chop the file up into list elements, one for each line. For small files (less than 1K) it doesn't really matter. For really large files (megabytes) you might induce paging with this approach.

Random access I/O

The `seek` and `tell` commands are used for random access to I/O streams. Each stream has a current position called the *seek offset*. Each read or write operation updates the seek offset by the number of bytes transferred. The current value of the offset is returned by the `tell` command. The `seek` command is used to set the seek offset by an amount, which can be positive or negative, from an origin, which is either start, current, or end.

Closing I/O streams

The close command is just as important as the others because it frees up operating system resources associated with the I/O stream. If you forget to close a stream it will be closed when your process exits. However, if you have a long-running program, like a Tk script, you might exhaust some O/S resources if you forget to close your I/O streams.

Note that the close command can raise an error. If the stream was a process pipeline and any of the processes wrote to their standard error stream, then this appears like an error to Tcl. The error is raised when the stream to the pipeline is finally closed. Similarly, if any of the processes in the pipeline exit with a non-zero status, close will raise an error.

The Current Directory - cd And pwd

The UNIX process has a current directory that is used as the starting point when resolving a relative pathname (a file name that does not begin with '/'). The pwd command returns the current directory, and the cd command is used to change the current directory. We'll use these commands in the example below that involves the glob command.

Matching File Names With glob

The glob command is used to expand a pattern into the set of matching file names. The pattern syntax is like that of the string match command in which * matches zero or more characters, ? matches a single character, and [*abc*] matches a set of characters. In addition, a file glob pattern can include a construct like {*a*,*b*,*c*} that will match any of *a*, *b*, or *c*. All other characters must match themselves. The general form of the glob command is:

```
glob ?flags? pattern ?pattern? ...
```

The -nocomplain flag causes glob to return an empty list if not files match the pattern. Otherwise glob will raise an error if no files match.

The -- flag is used to introduce the *pattern* if it begins with a -.

Unlike the glob matching in csh, the Tcl glob command only matches the names of existing files. (In csh, the {*a*,*b*} construct can match non-existent names.) In addition, the results of glob are not sorted. You'll have to use the lsort command to sort its result if that is important to you.

Example 7–12 Finding a file by name.

```
proc FindFile { startDir namePat } {
    set pwd [pwd]
```

```
    if [catch {cd $startDir} err] {
        puts stderr $err
        return
    }
    foreach match [glob -nocomplain -- $namePat]{
        puts stdout $startDir/$match
    }
    foreach file [glob -nocomplain *] {
        if [file isdirectory $file] {
            FindFile $startDir/$file $namePat
        }
    }
    cd $pwd
}
```

The FindFile procedure traverses the file system hierarchy using recursion. At each iteration it saves its current directory and then attempts to change to the next subdirectory . A catch is used to guard against bogus names. The glob command is used to match file names. FindFile is called recursively on each subdirectory .

The exit And pid commands

The exit command is used to terminate your script. Note that exit causes the whole UNIX process that was running the script to terminate. If you supply an integer -valued argument to exit then that becomes the exit status of the process.

The pid command returns the process ID of the current process. This can be useful as the seed for a random number generator because it will change each time you run your script. It is also common to embed the process ID in the name of temporary files.

CHAPTER 8

Reflection and Debugging

This chapter describes commands that give you a view into the interpreter. The `history` command and a simple debugger are useful during development and and debugging. The `info` command provides a variety of information about the internals of the Tcl interpreter. The `time` command measures the time it takes to execute a command.

Reflection provides feedback to a script about the internal state of the interpreter. This is useful in a variety of cases, from testing to see if a variable exists to dumping the state of the interpreter. This chapter starts with a description of the `info` command that provides lots of different information about the interpreter.

Interactive command history is the second topic of the chapter. The history facility can save you some typing if you spend a lot of time entering commands interactively.

Debugging is the last topic of the chapter. The old-fashioned approach of adding `puts` commands to your code is often quite useful. It takes so little time to add code and run another test that this is much less painful than if you had to wait for a long compilation everytime you changed a print command. The *tkinspect* program is an inspector that lets you look into the state of a Tk application. It can hook up to any Tk application dynamically, so it proves quite useful. Don Libes has implemented a Tcl debugger that lets you set breakpoints and step through your script. This debugger is described at the end of the chapter.

The info Command

Table 8–1 summarises the `info` command. The operations are described in more detail after the table.

Table 8–1 The `info` command.

`info args` *procedure*	A list of *procedure*'s arguments.
`info body` *procedure*	The commands in the body of *procedure*.
`info cmdcount`	The number of commands executed so far.
`info commands ?`*pattern*`?`	A list of all commands, or those matching *pattern*. Includes built-ins and Tcl procedures.
`info complete` *string*	True if *string* contains a complete Tcl command.
`info default` *proc arg var*	True if *arg* has a default parameter value in procedure *proc*. The default value is stored into *var*.
`info exists` *variable*	True if *variable* is defined.
`info globals ?`*pattern*`?`	A list of all global variables, or those matching *pattern*.
`info level`	The stack level of the current procedure, or 0 for the global scope.
`info level` *number*	A list of the command and its arguments at the specified level of the stack.
`info library`	The pathname of the Tcl library directory.
`info locals ?pattern?`	A list of t all local variables, or those matching *pattern*.
`info patchlevel`	The release patchlevel for Tcl.
`info procs ?pattern?`	A list of all Tcl procedures, or those that match *pattern*.
`info script`	The name of the file being processed, or NULL.
`info tclversion`	The version number of Tcl.
`info vars ?pattern?`	A list of all visible variables, or those matching pattern.

Variables

There are three categories of variables: local, global, and visible. Information about these categories is returned by the `locals`, `globals`, and `vars` operations, respectively. The local variables include procedure arguments as well as locally defined variables. The global variables include all variables defined at the global scope. The visible variables include locals, plus any variables made visible via `global` or `upvar` commands. Remember that a variable may not be defined yet even though a `global` command as declared it to belong to the global scope. Perhaps the most commonly used operation is `info exist`, to test whether a variable is defined or not.

A pattern can be specified to limit the returned list of variables to those that mach the pattern. The pattern is interpreted according to the rules of the `string match` command, which is described on page 24.

Procedures

You can find out everything about a Tcl procedure with the `args`, `body`, and `default` operations.This is illustrated in the `ShowProc` example given below. The puts commands use the `-nonewline` flag because the newlines in the procedure body, if any, are retained.

Example 8–1 Printing a procedure definition.

```
proc ShowProc {{namepat *} {file stdout}} {
    foreach proc [info procs $namepat] {
        set needspace 0
        puts -nonewline $file "proc $proc {"
        foreach arg [info args $proc] {
            if {$needspace) {
                puts -nonewline $file " "
            }
            if [info default $proc $arg value] {
                puts -nonewline $file "{$arg $value}"
            } else {
                puts -nonewline $file $arg
            }
        }
        # No newline needed because info body may return a
        # value that starts with a newline
        puts -nonewline $file "} {"
        puts -nonewline $file [info body $proc]
        puts $file "}"
    }
}
```

The `info commands` operation returns a list of all the commands, which includes both built-in commands defined in C and Tcl procedures. There is no operation that just returns the list of built-in commands. You have to write a procedure to take the difference of two lists to get that information.

The call stack

The `info level` operation returns information about the Tcl evaluation stack, or *call stack*. The global level is numbered zero. A procedure called from the global level is at level one in the call stack. A procedure it calls is at level two, and so on. The `info level` command returns the current level number of the stack if no level number is specified.

If a postitive level number is specified (e.g. `info level 3`) then the command returns the procedure name and argument values at that level in the call stack. If a negative level is specified, then it is relative to the current call stack. Relative level -1 is the level of the current procedure's caller, and relative-level 0 is the current procedure. The following example prints out the call stack. The CallTrace procedure avoids printing information about itself by starting at one less than the current call stack level. It prints a more descriptive header instead of its own call.

Example 8–2 Getting a trace of the Tcl call stack.

```
proc CallTrace {{file stdout}} {
    puts $file "Tcl Call Trace"
    for {set l [expr [info level]-1]} {$l > 0} {incr l -1} {
        puts $file "$l: [info level $l]"
    }
}
```

Command evaluation

The `info complete` operation figures out if a string is a complete Tcl command. This is useful for command interpreters that need to wait until the user has typed in a complete Tcl command before passing it to eval.

If you want to know how many Tcl commands are executed, use the `info cmdcount` command. This counts all commands, not just top-level commands. The counter is never reset, so you need to sample it before and after a test run if you want to know how many commands are executed during a test.

Scripts and the library

The name of the current script file is returned with the `info script` command. For example, if you use the `source` command to read commands from a file, then `info script` will return the name of that file if it is called during execution of the commands in that script. This is true even if the `info script` command is called from a procedure that is not defined in the script.

The pathname of the Tcl library is returned by the `info library` command. While you could put scripts into this directory, it might be better to have a separate directory and use the script library facility described in Chapter 9. This will make it easier to deal with new releases of Tcl, and to package up your code if you want other sites to use it.

Version numbers

Each Tcl release has a version number such as 7.4. This number is returned by the `info tclversion` command. If you want your script to run on a variety of Tcl releases, you may need to test the version number and take different actions in the case of incompatibilities between releases. If there are patches to the release, then a patch level is incremented. The patch level is reset to zero on each release, and it is returned by the `info tclpatchlevel` command.

Interactive Command History

The T cl shell programs keep a log of the commands that you type by using a history facility . The log is controlled and accessed via the `history` command. The history facility uses the term *event* to mean an entry in its history log. The events are just commands, but they have an event ID that is their index in the log. Y ou can also specify an event with a negative index that counts backwards from the end of the log. For example, event -1 is the previous event. T able 8–1 sumarises the T cl `history` command. Many forms take an event specifier , which defaults to -1.

Table 8–2 The `history` command.

`history`	Short for `history info` with no *count* .
`history add command ?exec?`	Add the command to the history list. If `exec` is specified, then execute the command.
`history change` *new* ?*event*?	Change the command specified by *event* to *new* in the command history .
`history event ?`*event*?	Returns the command specified by *event*.
`history info ?`*count*?	Returns a formatted history list of the last *count* commands, or of all commands.
`history keep` *count*	Limit the history to the last *count* commands.
`history nextid`	Returns the number of the next event.
`history redo ?`*event*?	Repeate the specified command.
`history substitute` *old new* ?*event*?	Globally replace *old* with *new* in the command specified by *event*, then execute the result.
`history words` *selector* ?*event*?	Return list elements from the event according to *selector*. List items count from zero. `$` is the last item. A range is specified as *a-b*, e.g., *1-$*.

In practice you will want to take advantage of the ability to abbreviate the history options and even the name of the `history` command itself. For the command you need to type a unique prefix, and this depends on what other commands are already defined. For the options, there are unique one-letter abbreviations for all of them. For example, you could reuse the last word of the previous command with `[hist w $]`. This works because a `$` that is not followed by alphanumerics (or an open brace) is treated as a literal `$`.

Several of the history operations update the history list. They remove the actual `history` command and replace it with the command that resulted from the history operation. The `event`, `redo`, `substitute`, and `words` operations all behave in this manner . This makes perfect sense because you'd rather have the actual command in the history instead of the history command used to retrieve the command.

History syntax

Some extra syntax is supported when running interactively to make the history facility more convenient to use. Table 8–1 shows the special history syntax supported by *tclsh* and *wish*.

Table 8–3 Special `history` syntax.

`!!`	Repeat the previous command.
`!`*`n`*	Repeat command number *`n`*. If *`n`* is negative it counts backward from the current command. The previous command is event -1.
`!`*`prefix`*	Repeat the last command that begins with *`prefix`*.
`!`*`pattern`*	Repeat the last command that matches *`pattern`*.
`^`*`old`*`^`*`new`*	Globally replace *`old`* with *`new`* in the last command.

The next example shows how some of the history operations work.

Example 8–3 Interactive `history` usage.

```
% set a 5
5
% set a [expr $a+7]
12
% history
   1 set a 5
   2 set a [expr $a+7]
   3 history
% !2
19
% !!
26
% ^7^13
39
% !h
   1 set a 5
   2 set a [expr $a+7]
   3 history
   4 set a [expr $a+7]
   5 set a [expr $a+7]
   6 set a [expr $a+13]
   7 history
```

A comparision to /bin/csh history syntax

The history syntax shown in the previous example is simpler than the history syntax provided by the C-shell. Not all of the history operations are supported with special syntax. The substitutions (using `^old^new`) are performed globally on the previous command. This is different than the quick-history of the C-shell. Instead, it is like the !:gs/old/new/ history command. So, for example, if the example had included ^a^b in an attempt to set b to 39, an error would have occurred because the command would have been changed to:

```
set b [expr $b+7]
```

If you want to improve the history syntax, you will need to modify the unknown command, which is where it is implemented. This command is discussed in more detail in Chapter 9. Here is the code from the unknown command that implements the extra history syntax. The main limitation in comparision withthe C-shell history syntax is that the ! substitutions are only performed when ! is at the beginning of the command.

Example 8–4 Implementing special history syntax.

```
# Excerpts from the standard unknown command
# uplevel is used to run the command in the right context
if {$name == "!!"} {
    return [uplevel {history redo}]
}
if [regexp {^!(.+)$} $name dummy event] {
    return [uplevel [list history redo $event]]
}
if [regexp {^\^([^^]*)\^([^^]*)\^?$} $name dummy old new] {
    return [uplevel [list history substitute $old $new]]
}
```

Debugging

The rapid turn around with Tcl coding means that it is often sufficient to add a few puts statements to your script to gain some insight about its behavior. This solution doesn't scale too well, however. A slight improvement is to add a Debug procedure that can have its output controlled better. You can log the information to a file, or turn it off completely. In a Tk application, it is simple to create a text widget to hold the contents of the log so you can view it from the application. Here is a simple Debug procedure. To enable it you need to set the debug(enable) variable. To have its output go to your terminal, set debug(file) to stderr.

Example 8–5 A Debug procedure.

```
proc Debug { string } {
    global debug
    if ![info exists debug(enabled)] {
        # Default is to do nothing
        return
    }
    puts $debug(file) $string
```

```
}
proc DebugOn {{file {}}} {
    global debug
    set debug(enabled) 1
    if {[string length $file] == 0} {
        if [catch {open /tmp/debug.out w} fileID] {
            put stderr "Cannot open /tmp/debug.out"
            set debug(file) stderr
        } else {
            puts stderr "Debug info to /tmp/debug.out"
            set debug(file) $fileID
        }
    }
}
proc DebugOff {} {
    global debug
    if [info exists debug(enabled)] {
        unset debug(enabled)
        flush $debug(file)
        if {$debug(file) != "stderr" &&
             $debug(file) != "stdout"} {
            close $debug(file)
            unset $debug(file)
        }
    }
}
```

Don Libes' debugger

Don Libes at the National Institute of Standards and Technology has built a Tcl debugger that lets you set breakpoints and step through your scripts interactively. He is also the author of the *expect* program that is described in Chapter 32. The debugger requires a modified Tcl shell because the debugger needs a few more built-in commands to support it. This section assumes you have it built into your shell already. The *expect* program includes the debugger, and creating a custom shell that includes the debugger is described in Chapter 32 on page 395.

The most interesting feature of the debugger is that you set breakpoints by specifying patterns that match commands. The reason for this is that Tcl doesn't keep around enough information to map from file line numbers to Tcl commands in scripts. The pattern matching is a clever alternative, and it opens up lots of possibilities.

The debugger defines several one-character command names. The commands are only defined when the debugger is active, and you shouldn't have one-letter commands of your own so it should not create any conflicts :-) The way you enter the debugger in the first place is left up to the application. The *expect* shell enters the debugger when you generate a keyboard interrupt, and Chapter 32 shows how you can set this up for a customized Tcl shell. Table 8–4 sumarises the debugger commands. They are described in more detail below.

Table 8–4 Debugger commands.

`s ?n?`	Step into a procedure. Step once, or *n* times.
`n ?n?`	Step over a procedure. Step over once, or *n* times.
`r`	Return from a procedure.
`b`	Set, clear or show a breakpoint.
`c`	Continue execution to next breakpoint or interrupt.
`w ?-w width? ?-c X?`	Show the call stack, limiting each line to *width* charcters. `-c` 1 displays control characters as escape sequences. -c 0 displays control characters normally.
`u ?level?`	Move scope up the call stack one level, or to level *level*.
`d ?level?`	Move scope down the call stack one level, or to level *level*.
`h`	Display help information.

When you are at the debugger prompt, you are talking to your Tcl interpreter so you can issue any Tcl command. There is no need to define new commands to look at variables. Just use `set`!

The `s` and `n` command are used to step through your script. They take an optional parameter that indicates how many steps to take before stopping again. The `r` command completes execution of the current procedure and stops right after the procedure returns.

The `w` command prints the call stack. Each level is preceeded by its number, with level 0 being the top of the stack. An asterisk is printed by the current scope, which you can change as described next. Each line of the stack trace can get quite long because of argument substitutions. Control the output width with the `-w` argument.

The `u` and `d` commands change the current scope. They move up and down the Tcl call stack, where "up" means towards the calling procedures. The very top of the stack is the global scope. You need to use these commands to easily examine variables in different scopes. They take an optional parameter that specifies what level to go to. If the level specifier begins with #, then it is an absolute level number and the current scope changes to that level. Otherwise the scope moves up or down the specified number of levels.

Breakpoints by pattern matching

The `b` command manipulates breakpoints. The location of a breakpoint is specified by a pattern. When a command is executed that matches the pattern, the breakpoint occurs. Eventually it will be possible to specify breakpoints by line number, but the Tcl interpreter doesn't keep around enough information to make that easy to do. The general form of the command to set a breakpoint is shown below.

```
b ?-re regexp? ?if condition? ?then action?
b ?-glob pattern? ?if condition? ?then action?
```

The `b` command supports both glob patterns and regular expressions. Patterns will be discussed in more detail below. A breakpoint can have a test associated with it. The breakpoint will only occur if the condition is met. A breakpoint can have an action, independent of a condition. The action provides a way to patch code into your script. Finally, the pattern itself is also optional, so you can have a breakpoint that is just a conditional. A breakpoint that just has an action will trigger on every command.

Here are several examples.

```
b -re ^foobar
```

This breaks whenever the `foobar` command is invoked. The `^` in the regular expression ensures that foobar is the first word in the command. In contrast, the next breakpoint occurs whenever foobar is about to be called from within another command. A glob pattern is used for comparision. A glob pattern has to match the whole command string, hence the asterisk at the beginning and end of the pattern.

```
b -glob {*\[foobar *}
```

The subpattern matching of the regular expression facility is supported. If you have subpatterns, the parts of the string that match are stored in the `dbg(1)` through `dbg(9)` array elements. The string that matched the whole pattern is stored in `dbg(0)`. The following breakpoint stops when the `crunch` command is about to be called with its first argument greater than `1024`.

```
b -re {^crunch ([0-9]+)} if {$dbg(1) > 1024}
```

If you just want to print information and keep going, you can put a `c`, `s`, `n`, or `r` command into the action associated with a breakpoint. The following breakpoint traces assignments to a variable.

```
b -re {^set a ([^ ]+)} then {
    puts "a changing from $a to $dbg(1)"
    c
}
```

The breakpoint is called before the command executes, so in this case `$a` refers to the old value, and the pattern extracts the new value. If an error occurs inside the action the error is discarded and the rest of the action is skipped.

Deleting break points

The b command with no arguments lists the defined breakpoints. Each breakpoint is preceeded by an ID number. To delete a breakpoint, give the breakpoint number preceeded by a minus sign:

```
b -N
```

The tkerror Command

When the Tk widgets encounter an error from a callback, such as the command associated with a button, they signal the error by calling the tkerror procedure. A default implementation displays a dialog and gives you an opportunity to view the Tcl call stack at the point of the error. You can supply your own version of tkerror. For example, my *exmh* application offers to send mail to me with a few words of explanation from the user and a copy of the traceback. I get interesting bug reports from all over the world!

The `tkerror` command is called with one argument that is the error message. The global variable `errorInfo` contains the stack trace information.

The tkinspect Program

The *tkinspect* program is a Tk application that lets you look at the state of other Tk applications. It displays procedures, variables, and the Tk widget hierarchy. With tkinspect you can issue commands to another application in order to change variables or test out commands. This turns out to be a very useful way to debug Tk applications. It was written by Sam Shen and is available in the Tcl archives. The current FTP address for this is:

```
ftp.aud.alcatel.com:/pub/tcl/code/tkinspect-4d.tar.gz
```

Performance Tuning

The `time` command measures the exectution time of a Tcl command. It takes an optional parameter that is a repetition count.

```
time {set a "Hello, World!"} 1000
=> 305 microseconds per iteration
```

This provides a very simple timing mechanism. A more advanced profiler is part of the Extended Tcl package, which is described on page 389. The profiler monitors the number of calls, the CPU time, and the elapsed time spent in different procedures.

Perhaps the most common performance bug in a Tcl program is the use of big lists instead of arrays. Extracting items from a list is expensive because the list must be reparsed in order to find each element. With a Tcl array you can access any element in constant time.

In general, iterating through large data structures is slow because Tcl reparses loop bodies and procedure bodies each time it evaluates them. Highly interative code is best optimized by moving it into C code.

If you really want to squeeze the last drop out of some Tcl code you can try shorting the names of variables and commands used in the inner loops. For reasons of good programming style you should not resort to this except in extreme cases. You can use the `rename` command to create short names for the commands used within a tight loop.

CHAPTER 9

Script Libraries

You can use a script library to collect useful Tcl procedures together so they can be used by more than one application. The library is implemented by the unknown command handler, which also provides a few other facilities. On of its features is the utomatic execution of UNIX programs instead of having to use the Tcl `exec` command.

Libraries are used to collect useful sets of Tcl procedures together so they can be used by multiple applications. For example, you could use any of the code examples that come with this book by creating a script library, and then directing your application to check in that library for missing procedures. One way to structure a large application is to have a short main script and a library of support scripts. The advantage of this approach is that not all the Tcl code needs to be loaded to get the application started. Then, as new features are accessed the code that implements them can be loaded.

If you are writing Tcl code that is designed to be used in a library, you need to pay attention to some coding conventions. Because there is no formal module system in Tcl, coding conventions have to be followed to avoid conflicts between procedures and global variables used in different packages. This chapter explains a simple coding convention for large Tcl programs.

The unknown Command

The Tcl library facility is made possible by the `unknown` command. Whenever the Tcl interpreter encounters a command that it does not know about, it calls the `unknown` command with the name of the missing command. The `unknown` command is implemented in Tcl, so you are free to provide your own mechanism to handle unknown commands. This chapter describes the behavior of the default implementation of `unknown`, which can be found in the init.tcl file in the Tcl library. The location of the library is returned by the `info library` command. In order to bootstrap the library facility, the Tcl shells (*tclsh* and *wish*) invoke the following Tcl command.

```
source [info library]/init.tcl
```

The tclIndex File

The unknown command uses an index to make the search for missing commands fast. When you create a script library, you will have to generate the index that records what procedures are defined in the library. The `auto_mkindex` procedure creates the index, which is stored in a file named `tclIndex` that is kept in the same directory as the files that make up the script library.

Suppose all the examples from this book are in the directory `/usr/local/tcl/welchbook`. You can make the examples into a script library just by creating the tclIndex file.

```
auto_mkindex /usr/local/tcl/welchbook *.tcl
```

You'll need to update the tclIndex file if you add procedures or change any of their names. A conservative approach to this is shown in the next example. It is conservative because it recreates the index if anything in the library has changed since the tclIndex file was last generated, whether or not the change added or removed a Tcl procedure.

Example 9–1 Maintaining a `tclIndex` file.

```
proc Library_UpdateIndex { libdir } {
    if ![file exists $libdir/tclIndex] {
        set doit 1
    } else {
        set age [file mtime $libdir/tclIndex]
        set doit 0
        foreach file [glob $libdir/*.tcl] {
            if {[file mtime $file] > $age} {
                set doit 1
                break
            }
        }
    }
    if { $doit } {
        auto_mkindex $libdir *.tcl
    }
}
```

Using A Library: auto_path

In order to use a script library you must inform the `unknown` command where to look. It uses the auto_path variable to record a list of directories to search for unknown commands. To continue our example, you can make the procedures in the book examples available by putting this command at the beginning of your scripts.

```
lappend auto_path /usr/local/tcl/welchbook
```

This has no effect if you have not created the tclIndex file. If you wanted to be extra careful you can do also call Library_UpdateIndex. This will update the index if you add new things to the library.

```
lappend auto_path /usr/local/tcl/welchbook
Library_UpdateIndex /usr/local/tcl/welchbook
```

This will not work if there is no tclIndex file at all because the unknown procedure won't be able to find the implementation of Library_UpdateIndex. Once the tclIndex has been created for the first time, then this will ensure that any new procedures added to the library will be installed into tclIndex. In practice, if you want this sort of automatic update it is wise to include something like the Library_UpdateIndex file directly into your application as opposed to loading it from the library it is supposed to be maintaining.

Disabling the library facility: auto_noload

If you do not want the unknown procedure to try and load procedures, you can set the auto_noload variable to disable the mechanism.

```
set auto_noload anything
```

How Auto Loading Works

If you look at the contents of a tclIndex file you will find that it defines an array named auto_index. One element of the array is defined for each procedure in the script library. The value of the array element is a command that will define the procedure. A line in the tclIndex file looks something like this.

```
set auto_index(Bind_Interface) "source $dir/bind_ui.tcl"
```

When the tclIndex file is read, the $dir gets substituted with the name of the directory that contains the tclIndex file, so the result is a source command that loads the file containing the Tcl procedure. The substitution is done with eval, so you could build a tclIndex file that contained any commands at all and count on $dir being defined properly. The next example is a simplified version of the code that reads the tclIndex file.

Example 9–2 Loading a tclIndex file.

```
# This is a simplified part of the auto_load
# command that processes a tclIndex file.
# Go throught auto_path from back to front
set i [expr [llength $auto_path]-1]
for {} {$i >= 0} {incr i -1} {
```

```
    set dir [lindex $auto_path $i]
    if [catch {open $dir/tclIndex]} f] {
        # No index
        continue
    }
    # eval the file as a script. Because eval is
    # used instead of source, an extra round of
    # substitutions is performed and $dir get expanded
    # The real code checks for errors here.
    eval [read $f]
    close $f
}
```

The behavior of the `auto_load` facility is exploited by schemes that dynamically link object code in order to define commands that are implemented in C. In those cases the `load` Tcl command is used. This is not a standard command, yet, because the details of dynamic linking vary considerably from system to system.

Interactive Conveniences

The `unknown` command provides a few other conveniences. These are only used when you are typing commands directly. They are disabled once execution enters a procedure or if the Tcl shell is not being used interactively. The convenience features are automatic execution of programs, command history, and command abbreviation. These options are tried, in order, if a command implementation cannot be loaded from a script library.

Auto Execute

The `unknown` procedure implements a second feature: automatic execution of external programs. This make a Tcl shell behave more like other UNIX shells that are used to execute programs. The search for external programs is done using the standard `PATH` environment variable that is used by other shells to find programs. If you want to disable the feature all together, set the `auto_noexec` variable.

```
set auto_noexec anything
```

History

The history facility described in Chapter 8 is implemented by the `unknown` procedure.

Abbreviations

If you type a unique prefix of a command then unknown will figure that out and execute the matching command for you. This is done after auto exec is attemped and history substitutions are performed.

Tcl Shell Library Environment

It may help to understand how the Tcl shells initialize their library environment. The first toehold on the environment is made when the shells are compiled. At that point the default pathname of the library directory is defined. For Tcl, this pathname is returned by the `info` command:

```
info library
```

For Tk, the pathname is defined by the `tk_library` variable *. One of the first things that a Tcl shell does is this:

```
source [info library]/init.tcl
```

The primary thing defined by `init.tcl` is the implementation of the `unknown` procedure. For Tk, `wish` also does this:

```
source $tk_library/tk.tcl
```

This initializes the scripts that support the Tk widgets. There are still more scripts, and they are organized as a library. So, the tk.tcl script sets up the auto_-path variable so the Tk script library is accessible. It does this:

```
lappend auto_path $tk_library
```

To summarize, the bootstrap works as follows:

- The Tcl C library defines the pathname returned by the `info library` command, and this default can be overridden with the `TCL_LIBRARY` environment variable.
- The Tcl interpreter sources `[info library]/init.tcl` in order to define the `unknown` command that implements the bulk of the library facility.
- The Tk C library defines a pathname and stores it into `tk_library`, a Tcl variable. The default can be overridden with the `Tk_LIBRARY` environment variable.
- The Tk interpreter sources `init.tcl` as above, and `$tk_library/tk.tcl`
- The Tk initialization script appends `$tk_library` to `auto_path`.

Normally these details are taken care of by the proper installation of the Tcl and Tk software, but I find it helps to understand things when you see all the steps in the initialization process.

Coding Style

If you supply a library then you need to follow some simple coding conventions to make your library easier to use by other programmers. The main problem is that there is no formal module system in Tcl, so you must follow some conventions to

* You can also override these settings with environment variables), `TCL_LIBRARY` and `TK_LIBRARY`, but you shouldn't have to resort to that.

avoid name conflicts with other library packages and the main application.

A module prefix for procedure names

The first convention is to choose an identifying prefix for the procedures in your package. For example, the preferences package in Chapter 28 uses `Pref` as its prefix. All the procedures provided by the library begin with `Pref`. This convention is extended to distinguish between private and exported procedures. An exported procedure has an underscore after its prefix, and it is OK to call this procedure from the main application or other library packages. Examples include `Pref_Add`, `Pref_Init`, and `Pref_Dialog`. A private procedure is meant for use only by the other procedures in the same package. Its name does not have the underscore. Examples include `PrefDialogItem` and `PrefXres`.

A global array for state variables

You should use the same prefix on the global variables used by your package. You can alter the capitalization, just keep the same prefix. I capitalize procedure names and use lowercase for variables. By sticking with the same prefix you identify what variables belong to the package and you avoid conflict with other packages.

In general I try to use a single global array for a package. The array provides a convenient place to collect together a set of related variables, much like a struct is used in C. For example, the preferences package uses the `pref` array to hold all its state information. It is also a good idea ot keep the use of the array private. It is better coding practice to provide exported procedures than to let other modules access your data structures directly. This makes it easier to change the implementation of your package without affecting its clients.

If you do need to export a few key variables from your module, use the underscore convention to distinquish exported variables too. If you need more than one global variable, just stick with the prefix convention to avoid conflicts.

If you are dissapointed by the lack of real modules in Tcl, then you should consider one of the object system extentions for Tcl. The [incr tcl] package described in Chapter 32 provides classes that have their own scope for member functions and instance variables.

C H A P T E R 10

Tk Fundamentals

This chapter introduces the basic concepts used in the Tk toolkit for the X window system. Tk adds about 35 Tcl commands that let you create and manipulate widgets in a graphical user interface.

Tk is a toolkit for window programming. It was been designed for the X window system, although ports to other window systems are expected to appear soon. Tk shares many concepts with other windowing toolkits, but you don't need to know much about graphical user interfaces to get started with Tk.

Tk provides a set of Tcl commands that create and manipulate *widgets*. A widget is a window in a graphical user interface that has a particular appearance and behavior. The terms *widget* and *window* are often used interchangeably. Widget types include buttons, scrollbars, menus, and text windows. Tk also has a general purpose drawing widget called a canvas that lets you create lighter-weight items like lines, boxes and bitmaps. The Tcl commands added by the Tk extension are summarized at the end of this chapter.

The X window system supports a hierarchy of windows, and this is reflected by the Tk commands, too. To an application, the window hierarchy means that there is a primary window, and then inside that window there can be a number of children windows. The children windows can contain more windows, and so on. Just as a hierarchical file system has directories that are containers for files and directories, a hierarchical window system uses windows as containers for other windows. The hierarchy affects the naming scheme used for Tk widgets as described below, and it is used to help arrange widgets on the screen.

Widgets are under the control of a *geometry manager* that controls their size and location on the screen. Until a geometry manager learns about a widget, it will not be mapped onto the screen and you will not see it. There are a few different geometry managers you can use in Tk, although this book primarily uses the *packer*. The main trick with any geometry manger is that you use *frame* widgets as containers for other widgets. One or more widgets are created and then arranged in a frame by a geometry manager. The packer is discussed in detail in Chapter 12.

A Tk-based application has an event driven control flow, just as with most window system toolkits. An event is handled by associating a Tcl command to that event using the `bind` command. There are a large number of different events defined by the X protocol, including mouse and keyboard events. Tk widgets have default bindings so you do not have to program every detail yourself. Bindings are discussed in detail in Chapter 13. You can also arrange for events to occur after a specified period of time with the `after` command. The event loop is implemented by the `wish` shell, or you can provide the event loop in your own C program as described in Chapter 29.

Event bindings are structured into a simple hierarchcy of global bindings, class bindings, and instance bindings. An example of a class is `Button`, which is all the button widgets. The Tk toolkit provides the default behavior for buttons as bindings on the `Button` class. You can supliment these bindings for an individual button, or define global bindings that apply to all bindings. You can even introduce new binding classes in order to group sets of bindings together. The binding hierarchy is controlled with the `bindtags` command.

A concept related to binding is *focus*. At any given time, one of the widgets has the input focus, and keyboard events are directed to it. There are two general approaches to focusing: give focus to the widget under the mouse, or explicitly set the focus to a particular widget. Tk provides commands to change focus so you can implement either style of focus management. To support modal dialog boxes, you can forcibly *grab* the focus away from other widgets. Chater 17 describes focus, grabs, and dialogs.

The basic structure of a Tk script begins by creating widgets and arranging them with a geometry manager, and then binding actions to the widgets. After the interpreter processes the commands that initialize the user interface, the event loop is entered and your application begins running.

If you use `wish` interactively, it will create and display an empty main window and give you a command line prompt. With this interface, your keyboard commands are handled by the event loop, so you can build up your Tk interface gradually. As we will see, you will be able to change virtually all aspects of your application interactively.

Hello World In Tk

Example 10–1 “Hello, World!” Tk program.

```
#!/usr/local/bin/wish -f
button .hello -text Hello \
```

```
    -command {puts stdout "Hello, World!"}
pack .hello -padx 20 -pady 10
```

This three-line script creates a button that prints a message when you click it. A picture of the interface is shown below. Above the button widget is a title bar that is provided by the window manager, which in this case is `twm`.

The first line identifies the interpreter for the script.

```
#!/usr/local/bin/wish -f
```

This special line is necessary if the script is in a file that will be used like other `UNIX` command files. The `-f` flag is required in versions of Tk before 4.0. Remember, on many `UNIX` systems the whole first line is limited to 32 characters, including the `#!` and the `-f`.

The `button` command creates an instance of a button.

```
button .hello -text Hello \
    -command {puts stdout "Hello, World!"}
=> .hello
```

The name of the button is `.hello`. The label on the button is `Hello`, and the command associated with the button is:

```
puts stdout "Hello, World!"
```

The pack command maps the button onto the screen. Some padding parameters are supplied so there is space around the button.

```
pack .hello -padx 20 -pady 10
```

If you type these two commands into `wish`, you won't see anything happen when the `button` command is given. After the `pack` command, though, you will see the empty main window shrink down to be just big enough to contain the button and its padding. The behavior of the packer will be discussed further in Chapter 11 and Chapter12.

Tk uses an object-based system for creating and naming widgets. Associated with each class of widget (e.g., Button) is a command that creates instances of that class of widget. As the widget is created, a new Tcl command is defined that operates on that instance of the widget. The example creates a button named `.hello`, and we can operate on the button using its name as a command. For example, we can cause the button to highlight a few times:

```
.hello flash
```

Or, we can run the command associated with the button:

```
.hello invoke
=> Hello, World!
```

Naming Tk Widgets

The period in the name of the button instance, `.hello`, is required. Tk uses a naming system for the widgets that reflects their position in a hierarchy of widgets. The root of the hierarchy is the main window of the application, and its name is simply `"."`. This is similar to the naming convention for directories in UNIX where the root directory is named `"/"`, and then `/` is used to separate components of a file name. Tk uses `"."` in the same way. Each widget that is a child of the main window is named something like `.foo`. A child widget of `.foo` would be `.foo.bar`, and so on. Just as file systems have directories that are containers for files (and other directories), the Tk window hierarchy uses frame widgets that are containers for widgets (and other frames).

There is one drawback to the Tk widget naming system. If your interface changes enough it can result in some widgets changing their position in the widget hierarchy, and hence having to change their name. You can insulate yourself from this programming nuisance by using variables to hold the names of important widgets. Use a variable reference instead of widget pathnames in case you have to change things, or in case you want to reuse your code in a different interface.

Configuring Tk Widgets

The example illustrates a style of named parameter passing that is prevalent in the Tk commands. Pairs of arguments are used to specify the attributes of a widget. The attribute names begin with a `-`, such as `-text`, and the next argument is the value of that attribute. Even the simplest Tk widget can have a dozen or more attributes that can be specified this way, and complex widgets can have 20 or more attributes. However, the beauty of Tk is that you only need to specify the attributes for which the default value is not good enough. This is illustrated by the simplicity of this `Hello, World` example.

Finally, each widget instance supports a `configure` (often abbreviated to `config`) operation that can query and change these attributes. The syntax for `config` uses the same named argument pairs used when you create the widget. For example, we can change the background color of the button to be red even after it has been created and mapped onto the screen.

```
.hello config -background red
```

You can use configure to query the current value of an attribute by leaving off the value. For example:

```
.hello config -background
=> -background background Background #ffe4c4 red
```

The returned information includes the command line switch, the resource name, the class name, the default value, and the current value, which is last. The class and resource name have to do with the X resource mechanism. In most cases you just need the current value, and you can use the `cget` operation for that.

```
.hello cget -background
=> red
```

Widgets attributes can be redefined any time, even the text and command that were set when the button was created. The following command changes `.hello` into a goodbye button:

```
.hello config -text Goodbye! -command exit
```

About The Tk Man Pages

The on-line manual pages that come with Tk provide a complete reference source for the Tk commands. You should be able to use the UNIX *man* program to read them.

```
% man button
```

There are a large number of attributes that are common across most of the Tk widgets. These are described in a separate man page under the name `options`. Each man page begins with a **`STANDARD OPTIONS`** section that lists which of these standard attributes apply, but you have to look at the `options` man page for the description.

Each attribute has three labels: its command-line switch, its name, and its class. The command-line switch is the format you use in Tcl scripts. This form is always all lowercase and prefixed with a hyphen (e.g., `-offvalue`).

The name and class have to do with X resource specifications. The resource name for the attribute has no leading hyphen, and it has uppercase letters at internal word boundaries (e.g., `offValue`). The resource class begins with an upper case letter and has uppercase letters at internal word boundaries. (e.g., `OffValue`). You need to know these naming conventions if you specify widget attributes via the X resource mechanism, which is described in more detail in Chapter 27. In addition, the tables in this book list widget attributes by their resource name because the command line switch can be derived from the resource name by mapping it to all lowercase.

The primary advantage to using resources to specify attributes is that you do not have to litter your code with attribute specifications. With just a few resource database entries you can specify attributes for all your widgets. In addition, if attributes are specified with resources, users can provide alternate resource specifications in order to override the values supplied by the application. For attributes like colors and fonts, this feature can be important to users.

Summary Of The Tk Commands

The following two tables list the Tcl commands added by Tk. The first table lists commands that create widgets. There are 15 different widgets in Tk, although 4 of them are variations on a button, and 5 are devoted to different flavors of text display. The second table lists commands that manipulate widgets and provide associated functions like input focus, event binding, and geometry management. The page number in the table is the primary reference for the command, and there are other references in the index.

Table 10–1 Tk widget-creation commands

Command	Pg.	Description
`button`	145	Create a command button.
`checkbutton`	149	Create a toggle button that is linked to a Tcl variable.
`radiobutton`	149	Create one of a set of radio buttons that are linked to one variable.
`menubutton`	153	Create a button that posts a menu.
`menu`	153	Create a menu.
`canvas`	227	Create a canvas, which supports lines, boxes, bitmaps, images, arcs, text, polygons, and embedded widgets.
`label`	165	Create a read-only, one-line text label.
`entry`	180	Create a one-line text entry widget.
`message`	167	Create a read-only, multi-line text message.
`listbox`	183	Create a line-oriented, scrolling text widget.
`text`	212	Create a general purpose text widget.
`scrollbar`	172	Create a scrollbar that can be linked to another widget.
`scale`	169	Create a scale widget that adjusts the value of a variable.
`frame`	163	Create a container widget that is used with geometry managers.
`toplevel`	163	Create a frame that is a new top level X window.

Table 10–2 Tk widget-manipulation commands

Command	Pg.	Description
`after`	259	Execute a command after a period of time elapses.
`bell`	176	Ring the X bell device.
`bind`	133	Bind a Tcl command to an X event.
`bindtags`	134	Create binding classes and control binding inheritence.
`clipboard`	255	Manipulate the X clipboard.
`destroy`	200	Delete a widget.
`fileevent`	260	Associate Tcl commands with file descriptors.

Table 10–2 Tk widget-manipulation commands

`focus`	195	Control the input focus.
`grab`	197	Steal the input focus from other widgets.
`image`	281	Create and manipulate images.
`lower`	132	Lower a window in the stacking order.
`option`	325	Access the Xresources database.
`pack`	130	The packer geometry manager.
`place`	130	The placer geometry manager.
`raise`	132	Raise a window in the stacking order.
`selection`	254	Manipulate the X PRIMARY selection.
`send`	261	Send a Tcl command to another Tk application.
`tk`	313	Query internal Tk state (e.g., the color model)
`tkerror`	87	Handler for background errors.
`tkwait`	198	Block awaiting an event.
`update`	200	Update the display by going through the event loop.
`winfo`	308	Query window state.
`wm`	303	Interact with the window manager.

CHAPTER 11

Tk by Example

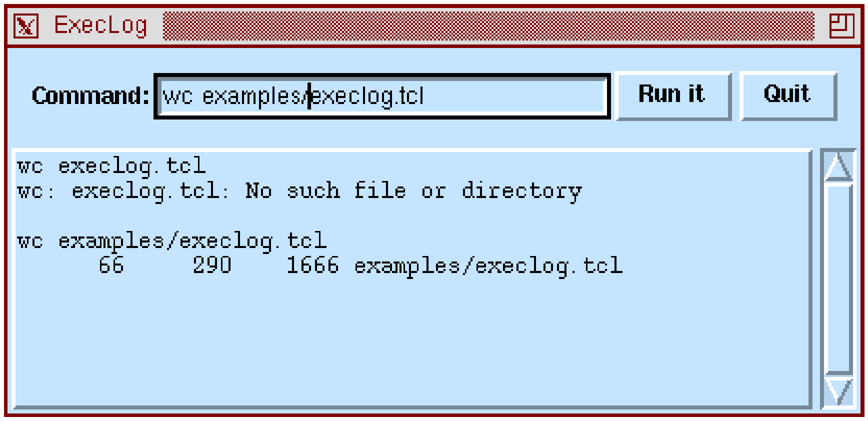
This chapter introduces Tk through a series of short examples.

Tk provides a quick and fun way to generate user interfaces. In this chapter we will go through a series of short example programs in order to give you a feel for what you can do. Some details are glossed over in this chapter and considered in more detail later. In particular, the packing geometry manager is covered in Chapter 12 and event bindings are discussed in Chapter 13. The Tk widgets are discussed in more detail in later chapters.

ExecLog

Our first example provides a simple user interface to running a `UNIX` program. The interface will consist of two buttons, `Run it` and `Quit`, an `entry` widget in which to enter a command, and a `text` widget in which to log the results of running the program. The script runs the program in a pipeline and uses the `fileevent` command to wait for output. This structure lets the user interface remain responsive while the program executes. You could use this to run *make*, for example, and it would save the results in the log. The complete example is given first, and then its commands are discussed in more detail.

Example 11–1 Logging the output of a UNIX program.




```
#!/usr/local/bin/wish -f
# execlog - run a UNIX program and log the output

# Set window title
wm title . ExecLog

# Create a frame for buttons and entry.
frame .top -borderwidth 10
pack .top -side top -fill x

# Create the command buttons.
button .top.quit -text Quit -command exit
set but [button .top.run -text "Run it" -command Run]
pack .top.quit .top.run -side right

# Create a labeled entry for the command
label .top.l -text Command: -padx 0
entry .top.cmd -width 20 -relief sunken \
   -textvariable command
pack .top.l -side left
pack .top.cmd -side left -fill x -expand true

# Set up key binding equivalents to the buttons
bind .top.cmd <Return> Run
bind .top.cmd <Control-c> Stop
focus .top.cmd

# Create a text widget to log the output
frame .t
set log [text .t.log -width 80 -height 10 \
   -borderwidth 2 -relief raised -setgrid true \
   -yscrollcommand {.t.scroll set}]
scrollbar .t.scroll -command {.t.log yview}
pack .t.scroll -side right -fill y
pack .t.log -side left -fill both -expand true
```

```
pack .t -side top -fill both -expand true

# Run the program and arrange to read its input
proc Run {} {
    global command input log but
    if [catch {open "|$command"} input] {
        $log insert end $input\n
    } else {
        fileevent $input readable Log
        $log insert end $command\n
        $but config -text Stop -command Stop
    }
}
# Read and log output from the program
proc Log {} {
    global input log
    if [eof $input] {
        Stop
    } else {
        gets $input line
        $log insert end $line\n
        $log see end
    }
}
# Stop the program and fix up the button
proc Stop {} {
    global input but
    catch {close $input}
    $but config -text "Run it" -command Run
}
```

Window title

The first command sets the title that will appear in the title bar implemented by the window manager. Recall that "`.`" is the name of the main window.

```
wm title . ExecLog
```

The `wm` command is used to communicate with the window manager. The window manager is the program that lets you open, close, and resize windows. It implements the title bar for the window and probably some small buttons to close or resize the window. (Different window managers look different - the figure shows a *twm* title bar.)

A frame for buttons, etc.

A frame is created to hold the widgets that appear along the top of the interface. The frame has a border to provide some space around the widgets:

```
frame .top -borderwidth 10
```

The frame is positioned in the main window. The default packing side is the top, so the `-side top` is redundant here, but used for clarity. The `-fill x` packing option will make the frame fill out to the whole width of the main window.

```
pack .top -side top -fill x
```

Command buttons

Two buttons are created: one to run the command, the other to quit the program. Their names, `.top.quit` and `.top.run`, imply that they are children of the `.top` frame. This affects the pack command, which positions widgets inside their parent by default.

```
button .buttons.quit -text Quit -command exit
set but [button .buttons.ping -text "Run it" \
    -command Run]
pack .buttons.quit .buttons.ping -side right
```

A label and an entry

The `label` and `entry` are also created as children of the top frame. The `label` is created with no padding in the X direction so it can be positioned right next to the `entry`. The size of the `entry` is specified in terms of characters. The relief attribute gives the `entry` some looks to set it apart visually on the display. The contents of the `entry` widget are associated with the Tcl variable `command`.

```
label .top.l -text Command: -padx 0
entry .top.cmd -width 20 -relief sunken \
    -textvariable command
```

The `label` and `entry` are positioned to the left inside the `.top` frame. The additional packing parameters to the `entry` allow it to expand is packing space and fill up that extra area with its display. The difference between packing space and display space is discussed in Chapter 12.

```
pack .top.l -side left
pack .top.cmd -side left -fill x -expand true
```

Key bindings and focus

Key bindings are set up for the `entry` widget that provide an additional way to invoke the functions of the application. The `bind` command associates a Tcl command with an X event in a particular widget. The `<Return>` event is generated when the user presses the `Return` key on the keyboard. The `<Control-c>` event is generated when the letter '`c`' is typed while the `Control` key is already held down. For the events to go to the `entry` widget, `.top.cmd`, input focus must be given to the widget. By default, an `entry` widget gets the focus when you click the left mouse button in it. The explicit focus command is helpful for users with the focus-follows-mouse model. As soon as the mouse is over the main window they can type into the `entry`.

```
bind .top.host <Return> Run
bind .top.cmd <Control-c> Stop
focus .top.cmd
```

A resizable text and scrollbar

A `text` widget is created and packed into a frame along with a `scrollbar`. The `scrollbar` is a separate widget in Tk, and it can be connected to a few different widgets using the same setup as is used here. The `text` widget's `yscrollcommand` is used to update the display of the `scrollbar` when the text widget is modified, and the `scrollbar` widget's `command` is used to scroll the associated widget when the user manipulates the scrollbar.

The `setgrid` attribute of the text widget is turned on. This has two effects. The most important is that it allows interactive resizing of the main window. By default, a Tk window is not resizable interactively, although it can always be resized under program control. The other effect of gridding is to restrict the resize so that only a whole number of lines and average sized characters can be displayed.

```
frame .t
set log [text .t.log -width 80 -height 10 \
    -borderwidth 2 -relief raised -setgrid true\
    -yscrollcommand {.t.scroll set}]
scrollbar .t.scroll -command {.t.log yview}
pack .t.scroll -side right -fill y
pack .t.log -side left -fill both -expand true
pack .t -side top -fill both -expand true
```

A side effect of creating a Tk widget is the creation of a new Tcl command that operates on that widget. The name of the Tcl command is the same as the Tk pathname of the widget. In this script, the text widget command, `.t.log`, will be needed in several places. However, it is a good idea to put the Tk pathname of an important widget into a variable because that pathname can change if you reorganize your user interface. The disadvantage of this is that you must declare the variable as a global inside procedures. The variable `log` is used for this purpose in this example to demonstrate this style.

The Run proc

The `Run` procedure starts the `UNIX` program specified in the command `entry`. That value is available via the global `command` variable because of the `textvariable` attribute of the entry. The command is run in a pipeline so that it executes in the background. The `catch` command is used to guard against bogus commands. The variable `input` will be set to an error message, or to the normal `open` return that is a file descriptor. A trick is used so that the error output of the program is captured. The program is started like this:

```
if [catch {open "|$command |& cat"} input] {
```

The leading | indicates that a pipeline is being created. If *cat* is not used like this, then the error output from the pipeline, if any, shows up as an error message when the pipeline is closed. In this example it turns out to be awkward to distinguish between errors generated from the program and errors generated because of the way the Stop procedure is implemented. Furthermore, some programs interleave output and error output, and you might want to see the error output in order instead of all at the end.

If the pipeline is opened successfully, then a callback is setup using the fileevent command. Whenever the pipeline generates output then the script can read data from it. The Log procedure is registered to be called whenever the pipeline is readable.

```
fileevent $input readable Log
```

The command (or the error message) is inserted into the log. This is done using the name of the text widget, which is stored in the log variable, as a Tcl command. The value of the command is appended to the log, and a newline is added so its output will appear on the next line.

```
$log insert end $command\n
```

The text widget's insert function takes two parameters: a *mark* and a string to insert at that mark. The symbolic mark end represents the end of the contents of the text widget.

The run button is changed into a stop button after the program is started. This avoids a cluttered interface and demonstrates the dynamic nature of a Tk interface. Again, because this button is used in a few different places in the script, its pathname has been stored in the variable but.

```
$but config -text Stop -command Stop
```

The Log procedure

The Log procedure is invoked whenever data can be read from the pipeline, and end-of-file has been reached. This condition is checked first, and the Stop procedure is called to clean things up. Otherwise, one line of data is read and inserted into the log. The text widget's see operation is used to position the view on the text so the new line is visible to the user.

```
if [eof $input] {
    Stop
} else {
    gets $input line
    $log insert end $line\n
    $log see end
}
```

The Stop procedure

The Stop procedure terminates the program by closing the pipeline. This results in a signal, SIGPIPE, being delivered to the program the next time it does a write to its standard output. The close is wrapped up with a catch. This suppresses the errors that can occur when the pipeline is closed prematurely on the process. Finally, the button is restored to its run state so that the user can run another command.

```
catch {close $input}
$but config -text "Run it" -command Run
```

In most cases, closing the pipeline is adequate to kill the job. If you really need more sophisticated control over another process, you should check out the *expect* Tcl extension, which is described briefly in Chapter 32 on page 391.

The Example Browser

The next example is an initial version of a browser for the code examples that appear in this book. The basic idea is to provide a menu that selects the examples, and a text window to display the examples. Because there are so many examples, a cascaded menu is set up to group the examples by the chapter in which they occur .

Example 11–2 A browser for the code examples in the book.

```
#!/project/tcl/src/brent/wish
# browse0.tcl --
#   Browser for the Tcl and Tk examples in the book.
#   Version 0

# The directory containing all the tcl files
set browse(dir) /tilde/welch/doc/tclbook/examples

# Set up the main display
wm minsize . 30 5
wm title . "Tcl Example Browser, v0"

frame .menubar
pack .menubar -fill x
button .menubar.quit -text Quit -command exit
pack .menubar.quit -side right

# A label identifies the current example
label .menubar.label -textvariable browse(current)
pack .menubar.label -side right -fill x -expand true

# Look through the .tcl files for the keywords
# that group the examples.
foreach f [glob $browse(dir)/*.tcl] {
    if [catch {open $f} in] {
        puts stderr "Cannot open $f: $in"
        continue
    }
    while {[gets $in line] >= 0} {
        if [regexp -nocase {^# ([^ ]+) chapter} $line \
                x keyword] {
            lappend examples($keyword) $f
            close $in
            break
        }
```

```
    }
}
# Create the menubutton and menu
menubutton .menubar.ex -text Examples -menu .menubar.ex.m
pack .menubar.ex -side left
set m [menu .menubar.ex.m]

# Create a cascaded menu for each group of examples
set i 0
foreach key [lsort [array names examples]] {
    $m add cascade -label $key -menu $m.sub$i
    set sub [menu $m.sub$i -tearoff 0]
    incr i
    foreach item [lsort $examples($key)] {
        $sub add command -label [file tail $item] \
            -command [list Browse $item]
    }
}

# Create the text to display the example
frame .body
text .body.t -setgrid true -width 80 -height 25 \
    -yscrollcommand {.body.s set}
scrollbar .body.s -command {.body.t yview} -orient vertical
pack .body.s -side left -fill y
pack .body.t -side right -fill both -expand true
pack .body -side top -fill both -expand true
set browse(text) .body.t

# Display a specified file. The label is updated to
# reflect what is displayed, and the text is left
# in a read-only mode after the example is inserted.
proc Browse { file } {
    global browse
    set browse(current) [file tail $file]
    set t $browse(text)
    $t config -state normal
    $t delete 1.0 end
    if [catch {open $file} in] {
        $t insert end $in
    } else {
        $t insert end [read $in]
        close $in
    }
    $t config -state disabled
}
```

More about resizing windows

This example uses the `wm minsize` command to put a constraint on the minimum size of the window. The arguments specify the minimum width and height. These values can be interpreted in two ways. By default they are pixel values. However, if an internal widget has enabled *geometry gridding*, then the dimensions are in grid units of that widget. In this case the text widget enables gridding with its `setgrid` attribute, so the minimum size of the window will be set so that the text window is at least 30 characters wide by 5 lines high.

```
wm minsize . 30 5
```

The other important side effect of setting the minimum size is that it enables interactive resizing of the window. Interactive resizing is also enabled if gridding is turned on by an interior widget, or if the maximum size is constrained with the `wm maxsize` command.

Managing global state

The example uses the `browse` array to collect its global variables. This makes it simpler to reference the state from inside procedures because only the array needs to be declared global. As the application grows over time and new features are added, that global command won't have to be adjusted. This style also serves to emphasize what variables are important.

The example uses the array to hold the name of the example directory (`dir`), the Tk pathname of the text display (`text`), and the name of the current file (`current`).

Searching through files

The browser searches the file system to determine what it can display. It uses `glob` to find all the Tcl files in the example directory. Each file is read one line at a time with `gets`, and then `regexp` is used to scan for keywords. The loop is repeated here for reference.

```
foreach f [glob $browse(dir)/*.tcl] {
    if [catch {open $f} in] {
        puts stderr "Cannot open $f: $in"
        continue
    }
    while {[gets $in line] >= 0} {
        if [regexp -nocase {^# ([^ ]+) chapter} $line \
                x keyword] {
            lappend examples($keyword) $f
            close $in
            break
        }
    }
}
```

The example files contain lines like this:

```
# Canvas chapter
```

The `regexp` picks out the keyword `Canvas` with the `([^ ]+)` part of the pattern, and this gets assigned to the `keyword` variable. The `x` variable gets assigned the value of the whole match, which is more than we are interested in. Once the keyword is found the file is closed and the next file is searched. At the end of the `foreach` loop the `examples` array has an element defined for each chapter keyword, and the value of each element is a list of the files that had examples for that chapter.

Cascaded menus

The values in the examples array are used to build up a cascaded menu structure. First a menubutton is created that will post the main menu. It is associated with the main menu with its menu attribute. The menu is created, and it must be a child of the menu button for the menu display to work properly.

```
menubutton .menubar.ex -text Examples \
    -menu .menubar.ex.m
set m [menu .menubar.ex.m]
```

For each example a `cascade` menu entry is added to the menu and the associated `menu` is defined. Once again, the submenu is defined as a child of the main menu. The submenu gets filled out with `command` entries that browse the file. Note the inconsistency with menu entries. Their text is defined with the `-label` option, not `-text`. Other than this they are much like buttons. Chapter 14 describes menus in more detail.

```
set i 0
foreach key [lsort [array names examples]] {
    $m add cascade -label $key -menu $m.sub$i
    set sub [menu $m.sub$i -tearoff 0]
    incr i
    foreach item [lsort $examples($key)] {
        $sub add command -label [file tail $item] \
            -command [list Browse $item]
    }
}
```

The Browse proc

The `Browse` procedure is fairly simple. It sets `browse(current)` to be the name of the file. This changes the main label because of its `textvariable` attribute that ties it to this variable. The `state` attribute of the text widget is manipulated so that the text is read-only after the text is inserted. You have to set the `state` to `normal` before inserting the text, otherwise the `insert` has no effect. Later enhancements to the browser will relax its read-only nature. Here are a few commands from the body of Browse.

```
global browse
set browse(current) [file tail $file]
$t config -state normal
$t insert end [read $in]
$t config -state disabled
```

A Tcl Shell

This section demonstrates the text widget with a simple Tcl shell application. Instead of using some other terminal emulator, it provides its own terminal environment using a `text` widget. You can use the Tcl shell as a sandbox in which to try out Tcl examples. The browser can too, by sending Tcl commands to the shell. Because the shell is a separate program, the browser is insulated from crashes. The shell and the browser are hooked together in Chapter 21.

Example 11–3 A Tcl shell in a text widget.

```
#!/project/tcl/src/brent/wish
# Simple evaluator. It executes Tcl in its own interpreter
# and it uses up the following identifiers.
# Tk widgets:
#   .eval - the frame around the text log
# Procedures:
#   _Eval - the main eval procedure
# Variables:
#   prompt - the command line prompt
#   _t - holds the ID of the text widget

# A frame, scrollbar, and text
frame .eval
set _t [text .eval.t -width 80 -height 20 \
   -yscrollcommand {.eval.s set}]
scrollbar .eval.s -command {.eval.t yview}
pack .eval.s -side left -fill y
pack .eval.t -side right -fill both -expand true
pack .eval -fill both -expand true

# Insert the prompt and initialize the limit mark
.eval.t insert insert "Tcl eval log\n"
set prompt "tcl> "
.eval.t insert insert $prompt
.eval.t mark set limit insert
.eval.t mark gravity limit left
focus .eval.t

# Key bindings that limit input and eval things
bind .eval.t <Return> { _Eval .eval.t ; break }
bind .eval.t <Any-Key> {
   if [%W compare insert < limit] {
      %W mark set insert end
   }
}
bindtags .eval.t {.eval.t Text all}

proc _Eval { t } {
   global prompt _debug
   set command [$t get limit end]
   if [info complete $command] {
      set err [catch {uplevel #0 $command} result]
```

```
            if {$_debug} {
             puts stdout "$err: $result\n"
            }
            $t insert insert \n$result\n
            $t insert insert $prompt
            $t see insert
            $t mark set limit insert
            return
        }
    }
```

Naming issues

This example uses some funny names for variables and procedures. This is a crude attempt to limit conflicts with the commands that you will type at the shell. The comments at the beginning explain what identifiers are used by this script. With a small amount of C programing you can easily introduce multiple Tcl interpreters into a single process to avoid problems like this. There have been some extensions published on the net that provide this capability at the Tcl level. (ref `minterp` extension)

Text marks and bindings

The shell uses a text *mark* and some extra bindings to ensure that users only type new text into the end of the text widget. The `limit` mark keeps track of the boundary between the read-only area and the editable area. The mark is used in two ways. First, the `_Eval` procedure looks at all the text between `limit` and `end` to see if it is a complete Tcl command. If it is, it evaluates it at the global scope using `uplevel #0`. Second, the `<Any-Key>` binding checks to see where the insert point is, and bounces it to the `end` if the user tries to input text before the `limit` mark. Chapter 18 describes the text widget and its mark facility in more detail.

CHAPTER 12

The Pack Geometry Manager

This chapter explores the `pack` geometry manager that is used to position widgets on the screen. The place geometry manager is also briefly described.

Geometry managers arrange widgets on the screen. There are a few different geometry managers, and you can use different ones to control different parts of your user interface. This book primarily uses the `pack` geometry manager, which is a constraint-based system. Tk also provides the `place` geometry manager, which is discussed briefly at the end of this chapter. Another interesting geometry manager is the `table` geometry manager provided as part of the `BLT` extension package, which is reviewed in Chapter 32.

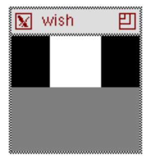
A geometry manager uses one widget as a parent, and it arranges multiple children (also called slaves) inside the parent. The parent is almost always a `frame`, but this is not strictly necessary. A widget can only be managed by one geometry manager at a time. If a widget is not managed, then it doesn't appear on your display at all.

The packer is a powerful constraint-based geometry manager. Instead of specifying in detail the placement of each window, the programmer defines some constraints about how windows should be positioned, and the packer works out the details. It is important to understand the algorithm used by the packer, otherwise the constraint-based results may not be what you expect.

This chapter explores the packer through a series of examples. We will start with a simple widget framework and then modify its layout to demonstrate how the packer works. The background of the main window is set to black, and the other frames are given different colors so you can identify frames and observe the effect of the different packing parameters.

Packing towards a side

Example 12–1 Two frames packed inside the main frame.



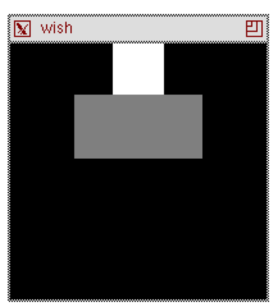

```
# Make the main window black
. config -bg black
# Create and pack two frames
frame .one -width 40 -height 40 -bg white
frame .two -width 100 -height 100 -bg grey50
pack .one .two -side top
```

The example creates two frames and packs them towards the top side of the main window. The upper frame, `.one`, is not as big and the main window shows through on either side. The children are packed towards the specified side in order, so `.one` is on top. The four possible side are `top`, `right`, `bottom`, and `left`. The `top` side is the default.

Shrinking frames and pack propagate

Note that the main window has shrunk down to be just large enough to hold its two children. In most cases this is the desired behavior. If it isn't you can turn it off with the `pack propagate` command. Apply this to the parent frame, and it will not adjust its size to fit its children.

Example 12–2 Turning off geometry propagation.




```
frame .one -width 40 -height 40 -bg white
```

```
frame .two -width 100 -height 100 -bg grey50
pack propagate . false
pack .one .two -side top
```

Horizontal And Vertical Stacking

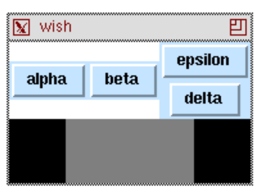
In general you should stick with either horizontal or vertical stacking within a frame. If you mix sides such as `left` and `top`, the effect might not be what you expect. Instead, you should introduce more frames to pack a set of widgets into a stack of a different orientation. For example, suppose we want to put a row of buttons inside the upper frame in the examples we have given so far.

Example 12–3 A horizontal stack inside a vertical stack.


```
frame .one -bg white
frame .two -width 100 -height 50 -bg grey50
# Create a row of buttons
foreach b {alpha beta gamma} {
    button .one.$b -text $b
    pack .one.$b -side left
}
pack .one .two -side top
```

You can build up more complex arrangements by introducing nested frames and switching between horizontal and vertical stacking as you go. Within each frame pack all the children with either a combination of `-side left` and `-side right`, or `-side top` and `-side bottom`.

Example 12–4 Even more nesting of horizontal and vertical stacks.




```
frame .one -bg white
frame .two -width 100 -height 50 -bg grey50
foreach b {alpha beta} {
    button .one.$b -text $b
    pack .one.$b -side left
}
```

```
# Create a frame for two more buttons
frame .one.right
foreach b {delta epsilon} {
   button .one.right.$b -text $b
   pack .one.right.$b -side bottom
}
pack .one.right -side right
pack .one .two -side top
```

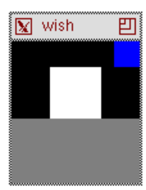
This example replaces the `.one.gamma` button with a vertical stack of two buttons, `.one.right.delta` and `.one.right.epsilon`. These are packed towards the bottom of `.one.right`, so the first one packed is on the bottom.

The frame `.one.right` was packed to the right, and in the previous example the button `.one.gamma` was packed to the `left`. In spite of the difference, they ended up in the same position relative to the other two widgets packed inside the `.one` frame. The next section explains why.

The Cavity Model

The packing algorithm is based on a *cavity model* for the available space inside a frame. For example, when the main `wish` window is created, the main frame is empty and there is an obvious space, or cavity, in which to place widgets. The primary rule about the cavity is that a widget occupies one whole side of the cavity. To demonstrate this, pack three widgets into the main frame. Put the first two on the bottom, and the third one on the right.

Example 12–5 Mixing bottom and right packing sides.




```
# pack two frames on the bottom.
frame .one -width 100 -height 50 -bg grey50
frame .two -width 40 -height 40 -bg white
pack .one .two -side bottom
# pack another frame to the right
frame .three -width 20 -height 20 -bg red
pack .three -side right
```

When we pack a third frame into the main window with `-side left` or `-side right`, then the new frame is positioned inside the cavity, which is above the two frames already packed toward the `bottom` side. The frame does not appear to the right of the existing frames as you might have expected. This is because the .two frame occupies the whole bottom side of the packing cavity, even though its display does not fill up that side.

Can you tell where the packing cavity is after this example? It is to the left of the frame .three, which is the last frame packed towards the right, and it is above the frame .two., which is the last frame packed towards the bottom. This explains why there was no difference between the previous two examples when .one.gamma was packed to the left side, but .one.right was packed to the right. At that point, packing to the left or right of the cavity had the same effect. However, it will affect what happens if another widget is packed into those two configurations. Try out the following commands after running Example 12–3 and Example 12–4 and compare the difference.*

```
button .one.omega -text omega
pack .one.omega -side right
```

Each packing parent has its own cavity, which is why introducing nested frames can help. If you stick with a horizontal or vertical arrangement inside any given frame, you can more easily simulate the packer's behavior in you head!

Packing Space and Display Space

The packer distinguishes between *packing* space and *display* space when it arranges the widgets. The display space is the area requested by a widget for the purposes of painting itself. The packing space is the area allowed by the packer for the placement of the widget. Because of geometry constraints, a widget may be allocated more (or less) packing space than it needs to display itself. The extra space, if any, is along the side of the cavity against which the widget was packed.

The -fill option

The -fill packing option causes a widget to fill up the allocated packing space with its display. A widget can fill in the X or Y direction, or both. The default is not to fill, which is why the black background of the main window has shown through in the examples so far.

Example 12–6 Filling the display into extra packing space.


```
frame .one -width 100 -height 50 -bg grey50
```

* Answer: After Example 12–3 the new button is to the right of all buttons. After Example 12–4 the new button is in between .one.beta and .one.right.

```
frame .two -width 40 -height 40 -bg white
# Pack with fill enabled
pack .one .two -side bottom -fill x
# pack another frame to the right
# The fill has no effect
frame .three -width 20 -height 20 -bg red
pack .three -side right -fill x
```

This is just like the previous example, except that `-fill x` has been specified for all the frames. The `.two` frame fills, but the `.three` frame does not. This is because the fill will not expand into the packing cavity. In fact, after this example, the packing cavity is the part that shows through in black. Another way to look at this is that the `.two` frame was allocated the whole bottom side of the packing cavity, so its fill can expand the frame to occupy that space. The `.three` frame has only been allocated the right side, so a fill in the x direction will not have any effect.

Another use of fill is for a menu bar that has buttons at either end and some empty space between them. The frame that holds the buttons is packed towards the top and fill is enabled in the X direction. Then, buttons can be packed into the left and right sides of the menubar frame. Without the fill, the menubar would shrink to be just large enough to hold all the buttons, and the buttons would be squeezed together.

Example 12–7 Using horizontal fill in a menubar.


```
frame .menubar -bg white
frame .body -width 150 -height 50 -bg grey50
# Create buttons at either end of the menubar
foreach b {alpha beta} {
    button .menubar.$b -text $b
}
pack .menubar.alpha -side left
pack .menubar.beta -side right
# Let the menu bar fill along the top
pack .menubar -side top -fill x
pack .body
```

Internal padding with -ipadx and -ipady

Another way to get more fill space is with the `-ipadx` and `-ipady` packing options that request more display space in the x and y directions, respectively. Due to other constraints the request might not be offered, but in general you can use this to give a widget more display space. The next example is just like the previous one except that some internal vertical padding has been added.

Example 12–8 The effects of internal padding (-ipady).

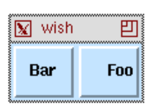

```
# Create and pack two frames
frame .menubar -bg white
frame .body -width 150 -height 50 -bg grey50
# Create buttons at either end of the menubar
foreach b {alpha beta} {
 button .menubar.$b -text $b
}
pack .menubar.alpha -side left -ipady 10
pack .menubar.beta -side right -ipadx 10
# Let the menu bar fill along the top
pack .menubar -side top -fill x -ipady 5
pack .body
```

The `alpha` button is taller and the `beta` button is wider because of the internal padding. With a frame the internal padding reduces the space available for the packing cavity, so the `.menubar` frame shows through above and below the buttons.

Some widgets have attributes that result in more display space. It would be hard to distinguish a frame with width `50` and no internal padding from a frame with width `40` and a `-ipadx 5` packing option. The packer would give the frame `5` more pixels of display space on either side for a total width of `50`.

Buttons have their own `-padx` and `-pady` options that give them more display space, too. The padding provided by the button is used to keep its text away from the edge of the button. The following example illustrates the difference. The `-anchor e` button option positions the text as far to the right as possible. Chapter 14 describes buttons and their attributes in more detail.

Example 12–9 Button padding vs. packer padding.

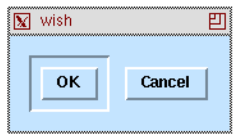



```
# Foo has internal padding from the packer
button .foo -text Foo -anchor e -padx 0 -pady 0
pack .foo -side right -ipadx 10 -ipady 10
# Bar has its own padding
button .bar -text Bar -anchor e -pady 10 -padx 10
pack .bar -side right -ipadx 0 -ipady 0
```

External padding with -padx and -pady

The packer can provide external padding that allocates packing space that cannot be filled. The space is outside of the border that widgets use to implement their 3D reliefs. Example 22–2 on page 273 shows the different reliefs. The look of a default button is achieved with an extra frame and some padding.

Example 12–10 The look of a default button.




```
. config -borderwidth 10
# OK is the default button
frame .ok -borderwidth 2 -relief sunken
button .ok.b -text OK
pack .ok.b -padx 5 -pady 5
# Cancel is not
button .cancel -text Cancel
pack .ok .cancel -side left -padx 5 -pady 5
```

Even if the `.ok.b` button were packed with `-fill both`, it would look the same. The external padding provided by the packer will not be filled by the child widgets.

Expand And Resizing

The `-expand true` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. Example 12–6 could use this on the small frame on top to get it to expand across the top of the display, even though it is packed to the right side. The more common case occurs when you have a resizable window. When the user makes the window larger, the widgets have to be told to take advantage of the extra space. Suppose you have a main widget like a `text`, `listbox`, or `canvas` that is in a `frame` with a `scrollbar`. That `frame` has to be told to expand into the extra space in its parent (e.g., the main window) and then the main widget (e.g., the `canvas`) has to be told to expand into its parent `frame`. Example 11–1 does this.

In nearly all cases the `-fill both` option is used along with `-expand true` so that the widget actually uses its extra packing space for its own display. The converse is not true. There are many cases where a widget should fill extra space, but not attempt to expand into the packing cavity. The examples below show the difference.

The main window can be made larger by interactive resizing, or under program control with the `wm geometry` command. By default interactive resizing is not enabled. You must use the `wm minisize` or `wm maxsize` commands which have the side effect of enabling interactive resizing. These commands place constraints on the size of the window. The `text`, `canvas`, and `listbox` widgets also have a `setgrid` attribute that, if enabled, makes the main window resizable. Chapter 24 describes geometry gridding.

Now we can investigate what happens when the window is made larger. The next example starts like Example 12–7 but the size of the main window is increased.

Example 12–11 Resizing without the expand option.


```
# Make the main window black
. config -bg black
# Create and pack two frames
frame .menubar -bg white
frame .body -width 150 -height 50 -bg grey50
# Create buttons at either end of the menubar
foreach b {alpha beta} {
    button .menubar.$b -text $b
}
pack .menubar.alpha -side left
pack .menubar.beta -side right
# Let the menu bar fill along the top
pack .menubar -side top -fill x
pack .body
# Resize the main window to be bigger
wm geometry . 200x100
# Allow interactive resizing
wm minsize . 100 50
```

The only widget that claims any of the new space is `.menubar` because of its `-fill x` packing option. The `.body` frame needs to be packed properly.

Example 12–12 Resizing with expand turned on.


```
# Use all of Example 12-11 then repack .body
```

```
pack .body -expand true -fill both
```

If more than one widget inside the same parent is allowed to expand, then the packer shares the extra space between them proportionally. This is probably not the effect you want in the examples we have built so far. The .menubar, for example is not a good candidate for expansion.

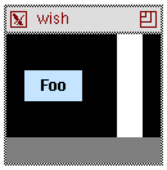
Example 12–13 More than one expanding widget.


```
# Use all of Example 12-11 then repack .menubar and .body
pack .menubar -expand true -fill x
pack .body -expand true -fill both
```

Anchoring

If a widget is left with more packing space than display space, then you can position within its packing space using the -anchor packing option. The default is -anchor center. The other options correspond to points on a compass: n, ne, e, se, s, sw, s, nw.

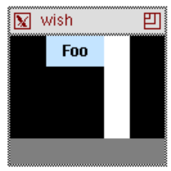
Example 12–14 Setup for anchor experiments.




```
# Make the main window black
. config -bg black
# Create two frames to hold open the cavity
frame .prop -bg white -height 80 -width 20
frame .base -width 120 -height 20 -bg grey50
pack .base -side bottom
# Float a label and the prop in the cavity
label .foo -text Foo
pack .prop .foo -side right -expand true
```

The .base frame is packed on the bottom. Then the .prop frame and the .foo label are packed to the right with expand set but no fill. Instead of being pressed up against the right side, the expand gives each of these widgets half of the extra space in the X direction. Their default anchor of `center` results in the position shown. The next example shows some different anchor positions.

Example 12–15 The effects of non-center anchors.




```
# Make the main window black
. config -bg black
# Create two frames to hold open the cavity
frame .prop -bg white -height 80 -width 20
frame .base -width 120 -height 20 -bg grey50
pack .base -side bottom
# Float the label and prop
# Change their position with anchors
label .foo -text Foo
pack .prop -side right -expand true -anchor sw
pack .foo -side right -expand true -anchor ne
```

The `label` has room on all sides, so each of the different anchors will position the it differently. The `.prop` frame only has room in the X direction, so it can only be moved into three different positions: left, center, and right. Any of the anchors `w`, `nw`, and `sw` will result in the left position. The anchors `center`, `n`, and `s` will result in the center position. The anchors `e`, `se`, and `ne` will result in the right position.

If you want to see all the variations, type in the following commands to animate the different packing anchors. The `update idletasks` forces any pending display operations. The `after 500` causes the script to wait for `500` milliseconds.

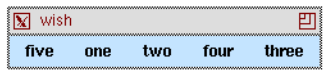
Example 12–16 Animating the packing anchors

```
foreach anchor {center n ne e se s sw w nw center} {
   pack .foo .prop -anchor $anchor
   # Update the display
   update idletasks
   # Wait half a second
   after 500
}
```

Packing Order

The packer maintains an order among the children that are packed into a frame. By default, each new child is appended to the end of the packing order. The most obvious effect of the order is that the children first in the packing order are closest to the side they are packed against. You can control the packing order with the `-before` and `-after` packing options, and you can reorganize widgets after they have already been packed.

Example 12–17 Controlling the packing order.




```
# Create five labels in order
foreach label {one two three four five} {
    label .$label -text $label
    pack .$label -side left
}
# ShuffleUp moves a widget to the beginning of the order
proc ShuffleUp { parent child } {
    set first [lindex [pack slaves $parent] 0]
    pack $child -in $parent -before $first
}
# ShuffleUp moves a widget to the end of the order
proc ShuffleDown { parent child } {
    pack $child -in $parent
}
ShuffleUp . .five
ShuffleDown . .three
```

pack slaves and pack info

The `pack slaves` command returns the list of children in their packing order. The `ShuffleUp` procedure uses this to find out the first child so it can insert another child before it. The `ShuffleDown` procedure is easier because the default is to append the child to the end of the packing order.

When a widget is repacked, then it retains all its packing parameters that have already been set. If you need to examine the current packing parameters for a widget use the `pack info` command.

```
pack info .five
=> -in . -anchor center -expand 0 -fill none -ipadx 0 \
     -ipady 0 -padx 0 -pady 0 -side left
```

Pack the scrollbar first

The packing order also determines what happens when the window is made too small. If the window is made small enough the packer will clip children that come later in the packing order. It is for this reason that when you pack a `scrollbar` and a `text` widget into a `frame`, pack the `scrollbar` first. Otherwise when the window is made smaller the `text` widget will take up all the space and the `scrollbar` will be clipped.

Choosing The Parent For Packing

In nearly all of the examples in this chapter a widget is packed into its parent frame. In general, it is possible to pack a widget into any descendent of its parent. For example, the `.a.b` widget could be packed into `.a`, `.a.c` or `.a.d.e.f`. The `-in` packing option lets you specify an alternate packing parent. One motivation for allowing this is that the frames introduced to get the arrangement right can cause cluttered names for important widgets. In Example 12–4 the buttons have names like `.one.alpha` and `.one.right.delta`, which is not that consistent. Here is an alternate implementation of the same example that simplifies the button pathnames and gives the same result.

Example 12–18 Packing into other relatives.

```
# Create and pack two frames
frame .one -bg white
frame .two -width 100 -height 50 -bg grey50
# Create a row of buttons
foreach b {alpha beta} {
    button .$b -text $b
    pack .$b -in .one -side left
}
# Create a frame for two more buttons
frame .one.right
foreach b {delta epsilon} {
    button .$b -text $b
    pack .$b -in .one.right -side bottom
}
pack .one.right -side right
pack .one .two -side top
```

When you do this, remember that the order in which you create widgets is important. Create the frames first, then create the widgets. The X stacking order for windows will cause the later windows to obscure the windows created first. The following is a common mistake.

```
button .a -text hello
frame .b
pack .a -in .b
```

If you cannot avoid this problem scenario, then you can use the `raise` command to fix things up.

```
raise .a
```

Unpacking a Widget

The `pack forget` command removes a widget from the packing order. The widget gets unmapped so it is not visible. A widget can only be managed by one parent, so you have to unpack it before moving the widget into another location. Unpacking a widget can also be useful if you want to suppress extra features of your interface. You can create all the parts of the interface, and just delay packing them in until the user requests to see them. If you unpack a parent frame, the packing structure inside it is maintained, but all the widgets inside the frame get unmapped.

Packer Summary

Keep these rules of thumb about the packer in mind.

- Pack vertically (`-side top` and `-side bottom`) or horizontally (`-side left` and `-side right`) within a frame. Only rarely will a different mixture of packing directions work out the way you want. Add frames in order to build up more complex structures.
- By default, the packer puts widgets into their parent frame, and the parent frame must be created before the children that will be packed into it.
- If you put widgets into other relatives, remember to create the frames first so the frames stay underneath the widgets packed into them.
- By default, the packer ignores `-width` and `-height` attributes of frames that have widgets packed inside them. It shrinks frames to be just big enough to allow for its borderwidth and to hold the widgets inside them. Use `pack propagate` to turn off the shrink wrap behavior.
- The packer distinguishes between packing space and display space. A widget's display might not take up all the packing space allocated to it.
- The `-fill` option causes the display to fill up the packing space in the x or y directions, or both.
- The `-expand true` option causes the packing space to expand into any room in the packing cavity that is otherwise unclaimed. If more than one widget in the same frame wants to expand, then they share the extra space.
- The `-ipadx` and `-ipady` options allocate more display space inside the border, if possible.
- The `-padx` and `-pady` options allocate more packing space outside the border, if possible.

The pack Command

Table 12–1 summarizes the `pack` command. Refer to the examples in the chapter for more detailed explanations of each command.

Table 12–1 A summary of the `pack` command.

`pack` *`win`* `?`*`win`* `..? ?`*`options`*`?`	This is just like `pack configure`.
`pack configure` *`win`* `?`*`win`* `...? ?`*`options`*`?`	Pack one or more widgets according to the *`options`*, which are given in the next table.
`pack forget` *`win`* `?`*`win`*`...?`	Unpack the specified windows.
`pack info` *`win`*	Return the packing parameters of *`win`*.
`pack propagate` *`win`* `?`*`bool`*`?`	Query or set the geometry propagation of *`win`*, which has other widgets packed inside it.
`pack slaves` *`win`*	Return the list of widgets managed by *`win`*.

Table 12–2 summarizes the packing options for a widget. These are set with the `pack configure` command, and the current settings are returned by the `pack info` command.

Table 12–2 Packing options.

`-after` *`win`*	Pack after *`win`* in the packing order.
`-anchor` *`anchor`*	`center n ne e se s sw s nw`
`-before` *`win`*	Pack before *`win`* in the packing order.
`-expand` *`boolean`*	Control expansion into the unclaimed packing cavity.
`-fill` *`style`*	Control fill of packing space. *`style`* is: `x y both none`
`-in` *`win`*	Pack inside *`win`*.
`-ipadx` *`amount`*	Horizontal internal padding, in screen units.
`-ipady` *`amount`*	Vertical internal padding, in screen units.
`-padx` *`amount`*	Horizontal external padding, in screen units.
`-pady` *`amount`*	Vertical external padding, in screen units.
`-side` *`side`*	`top right bottom left`

The Place Geometry Manager

The place geometry manager is much simpler than the packer. You specify the exact position and size of a window, or you specify the relative position and relative size of a widget. This is useful in a few situations, but it rapidly becomes tedious if you have to position lots of windows. The following `place` command centers a window in its parent. I use this command to position dialogs that I don't want to be detached top-level windows.

```
place $w -in $parent -relx 0.5 -rely 0.5 -anchor center
```

The `-relx` and `-rely` specify the relative X and Y position of the anchor point of the widget `$w` in `$parent`. The value is a number between zero and one, so `0.5` specifies the middle. The anchor point determines what point in `$w` is positioned according to the specifications. In this case the `center` anchor point is used so that the center of `$w` is centered in `$parent`. The default anchor point for windows is their upper-left hand corner (`nw`).

It is not necessary for `$parent` to actually be the parent widget of `$w`. The requirement is that `$parent` be the parent, or a descendent of the parent, of `$w`. It also has to be in the same toplevel window. This guarantees that `$w` is visible whenever `$parent` is visible. The following command positions a window five pixels above a sibling widget. If `$sibling` is repositioned, then `$w` moves with it.

```
place $w -in $sibling -relx 0.5 -y -5 -anchor s \
    -bordermode outside
```

The `-bordermode outside` option is specified so that any decorative border in `$sibling` is ignored when positioning `$w`. In this case the position is relative to the outside edge of `$sibling`. By default, the border is taken into account to make it easy to position widgets inside their parent's border.

You do not have to place a widget inside a frame, either. I use the first `place` command shown above to place a dialog in the middle of a `text` widget. In the second command, `$sibling` and $w might both be `label` widgets, for example.

The place Command

Table 12–1 summarizes the usage of the `place` command.

Table 12–3 A summary of the `place` command.

`place` *`win`* `?`*`win`* `..? ?`*`options`*`?`	This is just like `place configure`.
`place configure` *`win`* `?`*`win`* `...? ?`*`options`*`?`	Place one or more widgets according to the *`options`*, which are given in the next table.
`place forget` *`win`* `?`*`win`*`...?`	Unmap the windows
`place info` *`win`*	Return the placement parameters of *`win`*.
`place slaves` *`win`*	Return the list of widgets managed by *`win`*.

Table 12–4 summarizes the placement options for a widget. These are set with the `place configure` command, and the current settings are returned by the `place info` command.

Table 12–4 Placement options.

`-in` *win*	Place inside (or relative to) *win*.
`-anchor` *where*	`nw n ne e se s sw s center`. `nw` is the default.
`-x` *coord*	X position, in screen units, of the anchor point.
`-relx` *offset*	Relative X position. 0.0 is the left edge. 1.0 is the right edge.
`-y` *coord*	Y position, in screen units, of the anchor point.
`-rely` *offset*	Relative Y position. 0.0 is the top edge. 1.0 is the bottom edge.
`-width size`	Widget of the window, in screen units.
`-relwidth size`	Width relative to parent's width. 1.0 is full width.
`-height isze`	Height of the window, in screen units.
`-relheight size`	Height relative to the parent's height. 1.0 is full height.
`-bordermode mode`	If mode is `inside`, then size and position is inside the parent's border. If mode is `outside`, then size and position are relative to the outer edge of the parent.

Window Stacking Order

The `raise` and `lower` commands are used to control the X window stacking order. X has a window hierarchy, and the stacking order controls the relative position of sibling windows. By default, the stacking order is determined by the order that windows are created. Newer widgets are higher in the stacking order so they obscure older siblings. Consider this sequence of commands.

```
button .one
frame .two
pack .one -in .two
```

If you do this, you will not see the button. The problem is that the frame is higher in the stacking order so it obscures the button. You can change the stacking order with the `raise` command.

```
raise .one .two
```

This puts `.one` just above `.two` in the stacking order. If `.two` was not specified, then `.one` would be put at the top of the stacking order.

The `lower` command has a similar form. With one argument it puts that window at the bottom of the stacking order. Otherwise it puts it just below another window in the stacking order.

You can use `raise` and `lower` on toplevel windows to control their stacking order among all other toplevel windows. For example, if a user requests a dialog that is already displayed, use `raise` to make it pop to the foreground of their cluttered X desktop.

CHAPTER 13

Binding Commands to X Events

This chapter introduces the event binding mechanism in Tk. Bindings associated a Tcl command with an event like a mouse click or a key stroke.

Bindings associate a Tcl command with an event from the X window system. Events include key press, key release, button press, button release, mouse entering a window, mouse leaving, window changing size, window open, window close, focus in, focus out, and widget destroyed. These event types, and more, will be described in more detail in this chapter.

The bind Command

The `bind` command returns information about current bindings, and it defines new bindings. Called with a single argument, a widget or class identifier, `bind` returns the events for which there are command bindings.

```
bind Menubutton
=> <Key-Return> <Key-space> <ButtonRelease-1>
     <B1-Motion> <Motion> <Button-1> <Leave> <Enter>
```

These events are button-related events. `<Button-1>` for example, is the event generated when the user presses the first, or left-hand, mouse button. `<B1-Motion>` is a mouse motion event modified by the first mouse button. This event is generated when the user drags the mouse with the left button pressed. The event syntax will be described in more detail below.

If `bind` is given a key sequence argument then it returns the Tcl command bound to that sequence:

```
bind Menubutton <B1-Motion>
=> tkMbMotion %W down %X %Y
```

The Tcl commands in event bindings support an additional syntax for event keywords. These keywords begin with a percent and have one more character that identifies some attribute of the event. The keywords are replaced (i.e., substituted) with event-specific data before the Tcl command is evaluated. `%W` is replaced with the widget's pathname. The `%X and %Y` keywords are replaced with the coordinates of the event relative to the screen. The `%x` and `%x` keywords are replaced with the coordinates of the event relative to the widget. The event keywords are summarized below.

The `%` substitutions are performed throughout the entire command bound to an event, without regard to other quoting schemes. You have to use `%%` to obtain a single percent. For this reason you should make your binding commands short, adding a new procedure if necessary instead of littering percent signs throughout your code.

All, Class, And Widget Bindings

A hierarchy of binding information determines what happens when an event occurs. The default behavior of the Tk widgets are determined by class bindings. You can add bindings on a particular instance of a widget to supplement the class bindings. You can define global bindings by using the `all` keyword. The default ordering among bindings is to execute the global bindings first, then the class bindings, and finally the instance bindings.

Example 13–1 The binding hierarchy.

```
frame .one -width 30 -height 30
frame .two -width 30 -height 30
bind all <Control-c> {destroy %W}
bind all <Enter> {focus %W}
bind Frame <Enter> {%W config -bg red}
bind Frame <Leave> {%W config -bg white}
bind .two <Any-Button> {puts "Button %b at %x %y"}
focus default .
pack .one .two -side left
```

The example defines bindings at all three levels in the hierarchy. At the global level a handler for `<Control-c>` is defined. Because this is a keystroke, it is important to get the focus directed at the proper widget. Otherwise the main window has the focus, and the destroy command will destroy the entire application. In this case moving the mouse into a widget gives it the focus. If you prefer click-to-type, bind to `<Any-Button>` instead of `<Enter>`.

At the class level the `Frame` class is set up to change its appearance when the mouse moves into the window. At the instance level one of the frames is set up to report the location of mouse clicks.

The class for a widget is derived from the name of the command that creates it. A button widget has the class `Button`, a canvas has the class `Canvas`, and so on. You can define bindings for pseudo-classes as described below, which is useful for grouping bindings into different sets.

The bindtags command

The `bindtags` command controls the binding hierarchy, and with it you can specify one or more pseudo-classes as a source of bindings. One way to emulate the *vi* editor, for example, is to have two sets of bindings, one for insert mode and one for command mode.

```
bind InsertMode <Any-Key> {%W insert insert %A}
bind InsertMode <Escape> {bindtags %W {all CommandMode}}
bind CommandMode <Key-i> {bindtags %W {all InsertMode}}
```

Of course, you need to define many more bindings to fully implement all the *vi* commands. In this case the `bindtags` command has also simplified the binding hierarchy to include just global bindings and the mode-specific bindings. If it made sense, you could also reorder the hierarchy so that the global bindings were executed last, for example. The order that the tags appear in the `bindtags` command determines the order in which bindings are triggered.

break and continue in bindings

If you want to completely override the bindings for a particular widget you can use the `break` command inside the event handler. This stops the progression through the hierarchy of bindings. This works at any level, so a particular class could suppress global bindings.

The `continue` command in a binding stops the current binding and continues with the command from the next level in the binding hierarchy.

Note that you cannot use the break or continue commands inside a procedure that is called by the binding. This restriction is necessary because the procedure mechanism will not propogate the break. You would have to use the `return -code break` command to signal the break from within a procedure.

A note about bindings in earlier versions of Tk

In versions of Tk 3.6 and earlier, only one source of bindings for an event is used. If there is a binding on a widget instance for an event sequence, that binding overrides any class-specific or global bindings for that event sequence. Similarly, if there is a class-specific binding, then that overrides a global binding. You must be careful if you modify the bindings on a widget if you do not want to disable the default behavior. The following trick in Tk 3.6 (and earlier) ensures that the class-specific binding is executed before the new binding.

```
bind .list <Button-1> "[bind Listbox <Button-1>] ; Doit"
```

Event Syntax

The `bind` command uses the following syntax to describe events.

`<modifier-modifier-type-detail>`

The primary part of the description is the *type*, e.g. `Button` or `Motion`. The *detail* is used in some events to identify keys or buttons, .e.g. `Key-a` or `Button-1`. A *modifier* is another key or button that is already pressed when the event occurs, e.g., `Control-Key-a` or `B2-Motion`. There can be multiple modifiers, like `Control-Shift-x`.

The surrounding angle brackets are used to delimit a single event. The `bind` command allows a binding to a sequence of events, so some grouping syntax is needed. If there are no brackets, then the event defaults to a `KeyPress` event, and all the characters specify keys in a sequence. Sequences are described in more detail on page 141.

The following table briefly mentions all the event types. More information can be found on these events in the Event Reference section of the *Xlib Reference Manual*.

Table 13–1 Event types. Comma-separated types are equivalent.

`ButtonPress, Button`	A button is pressed (down).
`ButtonRelease`	A button is released (up).
`Circulate`	The window has had its stacking order change.
`Colormap`	The colormap has changed.
`Configure`	The window has changed size, position, border, or stacking order.
`Destroy`	The window has been destroyed.
`Enter`	The mouse has entered the window.
`Expose`	The window has been exposed.
`FocusIn`	The window has received focus.
`FocusOut`	The window has lost focus.
`Gravity`	The window has moved because of a change in size of its parent window.
`Keymap`	The keyboard mapping has changed.
`KeyPress, Key`	A key is pressed (down).
`KeyRelease`	A key is released (up).
`Motion`	The mouse is moving in the window.
`Leave`	The mouse is leaving the window.
`Map`	The window has been mapped (opened).

Table 13–1 Event types. Comma-separated types are equivalent.

`Property`	A property on the window has been changed or deleted.
`Reparent`	A window has been reparented.
`Unmap`	The window has been unmapped (iconified).
`Visibility`	The window has changed visibility.

The most commonly used events are key presses, button presses, and mouse motion. The `Enter` and `Leave` events indicate when the mouse enters a widget. The `Map` and `UnMap` events let an application respond to the open and close of the window. The `Configure` event is useful with a `canvas` if the display needs to be changed when the window resizes. The remaining events in the table have to do with dark corners of the X protocol, and they are seldom used.

Key Events

The `KeyPress` type is distinguished from `KeyRelease` so that you can have different bindings for each of these events. `KeyPress` can be abbreviated `Key`, and `Key` can be left off altogether if a detail is given to indicate what key. Finally, as a special case for KeyPress events, the angle brackets can also be left out. The following are all equivalent event specifications.

```
<KeyPress-a>
<Key-a>
<a>
a
```

The detail for a key is also known as the *keysym*, which is an X technical term that refers to the graphic printed on the key of the keyboard. For punctuation and non-printing characters, special keysyms are defined. Commonly encountered keysyms include (note capitalization):

```
Return, Escape, BackSpace, Tab, Up, Down, Left, Right,
comma, period, dollar, asciicircum, numbersign, exclam
```

The full list of definitions of these keysyms is buried inside an X11 header file, and it can also be affected by a dynamic keyboard map, the X modmap. You may find the next binding useful to determine just what the keysym for a particular key is on your system.*

```
bind $w <KeyPress> {puts stdout {%%K=%K %%A=%A}}
```

The `%K` keyword is replaced with the keysym from the event. The `%A` is replaced with the printing character that results from the event and any modifiers like `Shift`. The `%%` is replaced with a single percent sign. Note that these substitutions occur in spite of the curly braces used for grouping. If the user types a capital `Q`, the output is:

* Use <Any-KeyPress> in versions of Tk before 4.0 so that extra modifiers do not prevent the event from matching.

```
%K=Shift_R %A={}
%K=Q %A="Q"
```

In the first line with `%K=Shift_R` the {} indicates a `NULL`, a zero-valued byte, which is generated when modifier keys are pressed. The `NULL` can be detected in `<KeyPress>` bindings to avoid doing anything if only a modifier key is pressed. The following might be used with an entry widget.

```
bind $w <KeyPress> {
    if {%A != {}} {%W insert insert %A}
}
```

Button Events

Button events also distinguish between `ButtonPress`, (or `Button`), and `ButtonRelease`. `Button` can be left off if a detail specifies a button by number. The following are equivalent:

```
<ButtonPress-1>
<Button-1>
<1>
```

Note : the event `<1>` implies a `ButtonPress` event, while the event `1` implies a `KeyPress` event.

The mouse is tracked by binding to the `Enter`, `Leave`, and `Motion` events. `Enter` and `Leave` are triggered when the mouse comes into and exits out of the widget, respectively. A `Motion` event is generated when the mouse moves within a widget.

The coordinates of the mouse event are represented by the `%x` and `%y` keywords in the binding command. The coordinates are widget-relative, with the origin at the upper-left hand corner of a widget's window. The keywords `%X` and `%Y` represent the coordinates relative to the root window.

```
bind $w <Enter> {puts stdout "Entered %W at %x %y"}
bind $w <Leave> {puts stdout "Left %W at %x %y"}
bind $w <Motion> {puts stdout "%W %x %y"}
```

Other Events

The `<Map>` and `<Unmap>` events are generated when a window is opened and closed, or when a widget is packed or unpacked by its geometry manager.

The `<Configure>` event is generated when the window changes size. A canvas that computes its display based on its size can bind a redisplay procedure to the `<Configure>` event, for example. The `<Configure>` event can be caused by interactive resizing. It can also be caused by a `configure -width` widget command that changes the size of the widget. In general you should not reconfigure a widget's size while processing a `<Configure>` event to avoid an indefinite sequence of these events.

The `<Destroy>` event is generated when a widget is destroyed. (See also the description of the `wm` command. It is possible to register Tcl commands to handle various messages from the window manager.)

Chapter 17 presents some examples that use the `<FocusIn>` and `<FocusOut>` events.

Modifiers

A modifier indicates that another key or button is being held down at the time of the event. Typical modifiers are the `Shift` and `Control` keys. The mouse buttons can also be used as modifiers. If an event does not specify any modifiers, then the presence of a modifier key is ignored by the event dispatcher. However, if there are two possible matching events then the more accurate match will be used.

For example, consider these three bindings:

```
bind $w <KeyPress> {puts "key=%A"}
bind $w <Key-c> {puts "just a c"}
bind $w <Control-Key-c> {exit}
```

The last event is more specific than the others, and its binding will be triggered when the user types `c` with the `Control` key held down. If the user types `c` with the `Meta` key held down, then the second binding will be triggered. The `Meta` key is ignored because it doesn't match any binding. If the user types something other than a `c`, then the first binding will be triggered. If the user presses the `Shift` key, then the keysym that is generated will be `C`, not `c`, so the last two events will not match.

There are 8 modifier keys defined by the X protocol. The `Control`, `Shift`, and `Lock` modifiers are found on nearly all keyboards. The `Meta` and `Alt` modifiers tend to vary from system to system, and they may not be defined at all. They are commonly mapped to be the same as `Mod1` or `Mod2`, and Tk will try to determine how things are set up for you. The remaining modifiers, `Mod3` through `Mod5`, are sometimes mapped to other special keys. Table 13–2 summarizes the modifiers.

Table 13–2 Event modifiers.

`Control`	The Control key.
`Shift`	The shift key.
`Lock`	The caps-lock key.
`Meta, M`	Defined to be the modifier (`M1` through `M5`) that is mapped to the `Meta_L` and `Meta_R` keysyms.
`Alt`	Defined to be the modifier mapped to `Alt_L` and `Alt_R`.
`Mod1, M1`	The first modifier.
`Mod2,M2,Alt`	The second modifier.

Table 13–2 Event modifiers.

`Mod3,M3`	Another modifier .
`Mod4, M4`	Another modifier .
`Mod5, M5`	Another modifier .
`Button1, B1`	The first mouse button (left).
`Button2, B2`	The second mouse button (middle).
`Button3, B3`	The third mouse button (right).
`Button4, B4`	The fourth mouse button.
`Button5, B5`	The fifth mouse button.
`Double`	Matches double press event.
`Triple`	Matches triple press event.
`Any`	Matches any combination of modifiers.

The UNIX *xmodmap* program will return the current mappings from keys to these modifiers. The first column of its output lists the modifier . The rest of each line identifies the keysym(s) and low-level keycodes that are mapped to each modifier . The *xmodmap* program can also be used to change the mapping.

Example 13–2 Output from the `UNIX` *xmodmap* program.

```
xmodmap: up to 3 keys per modifier,
       (keycodes in parentheses):
shift Shift_L (0x6a), Shift_R (0x75)
lock Caps_Lock (0x7e)
control Control_L (0x53)
mod1 Meta_L (0x7f), Meta_R (0x81)
mod2 Mode_switch (0x14)
mod3 Num_Lock (0x69)
mod4 Alt_L (0x1a)
mod5 F13 (0x20), F18 (0x50), F20 (0x68)
```

The button modifiers, `B1` through `B5`, are most commonly used with the `Motion` event to distinguish different mouse dragging operations.

The `Double` and `Triple` events match on repetitions of an event within a short period of time. These are commonly used with mouse events. The main thing to be careful with is that the binding for the regular press event will match on the first press of the `Double`. Then the command bound to the `Double` event will match on the second press. Similarly , a `Double` event will match on the first two presses of a `Triple` event. V erify this by trying out the following bindings:

```
bind . <1> {puts stdout 1}
bind . <Double-1> {puts stdout 2}
bind . <Triple-1> {puts stdout 3}
```

Your bindings have to take into consideration that more than one command could result from a `Double` or `Triple` event. This effect is compatible with an interface that selects an object with the first click, and then operates on the selected object with a `Double` event. In an editor, character, word, and line selection on a single, double and triple click, respectively, is a good example.*

Events in Tk 3.6 and earlier

In earlier versions of Tk, before version 4.0, extra modifier keys prevented events from matching. If you wanted your bindings to be liberal about what modifiers were in effect, you had to use the `Any` modifier. This modifier is a wild card that matches if zero or more modifiers are in effect. You can still use `Any` in Tk 4.0 scripts, but it has no effect.

Event Sequences

The `bind` command accepts a sequence of events in a specification, and most commonly this is a sequence of key events.

```
bind . a {puts stdout A}
bind . abc {puts stdout C}
```

With these bindings in effect, both bindings will be executed when the user types `abc`. The binding for `a` will be executed when `a` is pressed, even though this event is also part of a longer sequence. This is similar to the behavior with `Double` and `Triple` event modifiers. For this reason you have to be careful when binding sequences. One trick is to put a null binding on the keypress used as the prefix of a command sequence.

```
bind $w <Control-x> { }
bind $w <Control-x><Control-s> Save
bind $w <Control-x><Control-c> Quit
```

The null command for `<Control-x>` ensures that nothing happens until the command sequence is completed. This trick is embodied by `BindSequence` in the next example. If a sequence is detected, then a null binding is added for the prefix. The procedure also supports the *emacs* convention that `<Meta-x>` is equivalent to `<Escape>x`. This convention arose because `Meta` is not that standard across keyboards. The `regexp` command is used to pick out the detail from the `<Meta>` event.

Example 13–3 Emacs-like binding convention for Meta and Escape.

```
proc BindSequence { w seq cmd } {
    bind $w $seq $cmd
```

* If you really want to disable this, you can experiment with using `after` to postpone processing of one event The time constant in the bind implementation of <Double> is 500 milliseconds. At the single click event, schedule its action to occur after 600 milliseconds, and verify at that time that the <Double> event has not occurred.

```
    # Double-bind Meta-key and Escape-key
    if [regexp {<Meta-(.*)>} $seq match letter] {
        bind $w <Escape><$letter> $cmd
    }
    # Make leading keystroke harmless
    if [regexp {(<.+>)<.+>} $seq match prefix] {
        bind $w $prefix { }
    }
}
```

Event Keywords

The keyword substitutions are described in the table below. Remember that these substitutions occur throughout the command, regardless of other Tcl quoting conventions. Keep your binding commands short, introducing procedures if needed. For the details about various event fields, consult the *Xlib Reference Manual*. The string values for the keyword substitutions are listed after a short description of the keyword. If no string values are listed, the keyword has an integer value like a coordinate or window ID. The events applicable to the keyword are listed last, in parentheses.

Table 13–3 A summary of the event keywords.

`%%`	Use this to get a single percent sign.
`%#`	The serial number for the event.
`%a`	The above field from the event. (`Configure`)
`%b`	Button number. (`ButtonPress`, `ButtonRelease`)
`%c`	The count field. (`Expose`, `Map`)
`%d`	The detail field. `NotifyAncestor`, `NotifyNonlinearVirtual`, `NotifyDetailNone`, `NotifyPointer`, `NotifyInferior`, `NotifyPointerRoot`, `NotifyNonlinear`, `NotifyVirtual`. (`Enter`, `Leave`, `FocusIn`, `FocusOut`)
`%f`	The focus field (0 or 1). (`Enter`, `Leave`)
`%h`	The height field. (`Configure`, `Expose`)
`%k`	The keycode field. (`KeyPress`, `KeyRelease`)
`%m`	The mode field. `NotifyNormal`, `NotifyGrab`, `NotifyUngrab`, `NotifyWhileGrabbed`. (`Enter`, `Leave`, `FocusIn`, `FocusOut`)
`%o`	The override_redirect field. (`Map`, `Reparent`, `Configure`)
`%p`	The place field. `PlaceOnTop`, `PlaceOnBottom`. (`Circulate`,)

Table 13–3 A summary of the event keywords.

`%s`	The state field. A decimal string. (`ButtonPress`, `ButtonRelease`, `Enter`, `Leave`, `KeyPress`, `KeyRelease`, `Motion`) `VisibilityUnobscured`, `VisibilityPartiallyObscured`, `VisibilityFullyObscured`. (`Visibility`)
`%t`	The time field.
`%v`	The value_mask field. (`Configure`)
`%w`	The width field. (`Configure`, `Expose`)
`%x`	The x coordinate, widget relative.
`%y`	The y coordinate, widget relative.
`%A`	The `ASCII` character from the event, or `NULL`. (`KeyPress`, `KeyRelease`)
`%B`	The border_width field. (`Configure`)
`%D`	The display field.
`%E`	The send_event field.
`%K`	The keysym from the event. (`KeyPress`, `KeyRelease`)
`%N`	The keysym as a decimal number . (`KeyPress`, `KeyRelease`)
`%R`	The root window ID.
`%S`	The subwindow ID.
`%T`	The type field.
`%W`	The Tk pathname of the widget receiving the event.
`%X`	The x_root field. Relative to the (virtual) root window . (`ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `Motion`)
`%Y`	The y_root field. Relative to the (virtual) root window . (`ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `Motion`)

CHAPTER 14

Buttons and Menus

Buttons and menus are the primary way that applications expose functions to users. This chapter describes how to create and manipulate buttons and menus.

A `button` is a classic Tk widget because it is associated with a Tcl command that invokes an action in the application. The `checkbutton` and `radiobutton` widgets affect an application indirectly by controlling a Tcl variable. A `menu` elaborates on this concept by organizing button-like items into related sets, including cascaded menus. The `menubutton` widget is a special kind of button that displays a menu when you click on it.

Associating a command to a button is often quite simple, as illustrated by the Tk Hello World example:

```
button .hello -command {puts stdout "Hello, World!}
```

This chapter describes a few useful techniques for setting up the commands in more general cases. If you use variables inside button commands, you have to understand the scoping rules that apply. This is the first topic of the chapter. Once you get scoping figured out, then the other aspects of buttons and menus are quite straight-forward.

Button Commands and Scope Issues

Perhaps the trickiest issue with button commands has to do with variable scoping. A button command is executed at the global scope, which is outside of any procedure. If you create a button while inside a procedure, then the button command will execute in a different scope later. This can be a source of confusion. A related issue is that when you define a button you may want the values of some variables as they are when the button is defined, while you want the value of other variables as they are when the button is used. I think of this as the "now" and "later" scopes. Again, when these two "scopes" are mixed, it can be confusing.

The next example illustrates the problem. The button command is an expression that includes the variable `x` that is defined in the global scope, and `val` that is defined locally. This mixture makes things awkward.

Example 14–1 A troublesome `button` command.


```
proc Trouble {args} {
    set b 0
    label .label -textvariable x
    set f [frame .buttons -borderwidth 10]
    foreach val $args {
        button $f.$b -text $val \
            -command "set x \[expr \$x * $val\]"
        pack $f.$b -side left
        incr b
    }
    pack .label $f
}
set x 1
Trouble -1 4 7 36
```

The example uses a label widget to display the current value of `x`. The `textvariable` attribute is used so that the `label` displays the current value of the variable, which is always a global variable. The button's command is executed at the global scope, so it updates the global variable `x`.

The definition of the `button` command is ugly, though. The value of the loop variable `val` is needed when the button is defined, but the rest of the substitutions need to be deferred until later. The variable substitution of `$x` and the command substitution of `expr` are suppressed by quoting with backslashes.

```
set x \[expr \$x * $val\]
```

In contrast, the following command will assign a constant expression to `x` each time the button is clicked, and it depends on the current value of `x`, which not defined in the version of `Trouble` above:

```
button $f.$b -text $val \
```

```
-command "set x [expr $x * $val]"
```

Another incorrect approach is to quote the whole command with braces. This defers too much, preventing the value of `val` from being used at the correct time.

The general technique for dealing with these sorts of scoping problems is to introduce Tcl procedures for use as the button commands. The troublesome example given above can be cleaned up by introducing a little procedure to encapsulate the expression.

Example 14–2 Fixing up the troublesome situation.

```
proc LessTrouble { args } {
    set b 0
    label .label -textvariable x
    set f [frame .buttons -borderwidth 10]
    foreach val $args {
        button $f.$b -text $val \
            -command "UpdateX $val"
        pack $f.$b -side left
        incr b
    }
    pack .label $f
}
proc UpdateX { val } {
    global x
    set x [expr $x * $val]
}
set x 1
LessTrouble -1 4 7 36
```

It may seem just like extra work to introduce the helper procedure, `UpdateX`. However, it makes the code clearer in two ways. First, you do not have to struggle with backslashes to get the button command defined correctly. Second, the code is much clearer about the function of the button. It's job is to update the global variable `x`.

Double quotes are used in the button command to allow $val to be substituted. Whenever you use quotes like this, you have to be aware of the possible values for the substitutions. If you are not careful, the command you create may not be parsed correctly. The safest way to generate the command is with the list procedure:

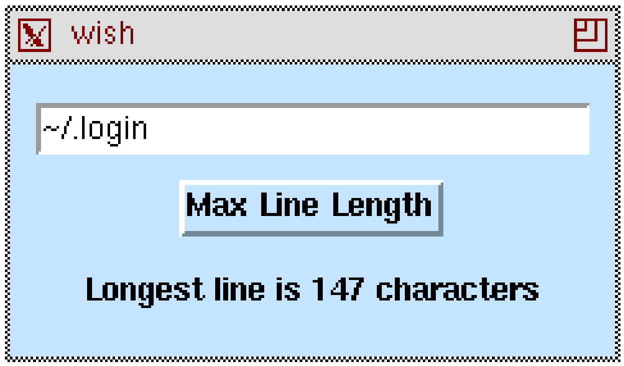
```
button $f.$b -text $val -command [list UpdateX $val]
```

The use of `list` ensures that the command is a list of two elements, `UpdateX` and the value of `val`. This is important because `UpdateX` only takes a single argument. If `val` contained white space then the resulting command would be parsed into more words than you expected. Of course, in this case we plan to always call LessTrouble with a set of numbers, which do not contain white space.

The next example provides a more straight-forward application of procedures for button commands. In this case the advantage of the procedure `Max-`

`LineLength` is that it creates a scope for the local variables used during the button action. This ensures that the local variables do not accidentally conflict with global variables used elsewhere in the program. There is also the standard advantage of a procedure, which is that you may find another use for the action in another part of your program.

Example 14–3 A `button` associated with a Tcl procedure.




```
proc MaxLineLength { file } {
    set max 0
    if [catch {open $file} in] {
        return $in
    }
    foreach line [split [read $in] \n] {
        set len [string length $line]
        if {$len > $max} {
            set max $len
        }
    }
    return “Longest line is $max characters”
}
# Create an entry to accept the file name,
# a label to display the result
# and a button to invoke the action
. config -borderwidth 10
entry .e -width 30 -bg white -relief sunken
button .doit -text “Max Line Length” \
    -command {.label config -text [MaxLineLength [.e get]]}
label .label -text “Enter file name”
pack .e .doit .label -side top -pady 5
```

The example is centered around the `MaxLineLength` procedure. This opens a file and loops over the lines finding the longest one. The file `open` is protected with `catch` in case the user enters a bogus file name. In that case, the procedure returns the error message from `open`. Otherwise the procedure returns a message about the longest line in the file. The local variables `in`, `max`, and `len` are hidden inside the scope of the procedure.

The user interface has three widgets, an `entry` for user input, the `button`, and a `label` to display the result. These are packed into a vertical stack, and the main window is given a borderwidth so things look OK. Obviously this simple UI can be improved in several ways. There is no `Quit` button, for example.

All the action happens in the button command:

```
.label config -text [MaxLineLength [.e get]]
```

Braces are used when defining the button command so that the command substitutions all happen when the button is clicked. The value of the `entry` widget is obtained with `.e get`. This value is passed into `MaxLineLength`, and the result is configured as the text for the `label`. This command is still a little complex for a button command. For example, suppose you wanted to invoke the same command when the user pressed `<Return>` in the entry. You would end up repeating this command in the entry binding. It might be better to introduce a one-line procedure to capture this action so it is easy to bind the action to more than one user action. Here is how that might look:

```
proc Doit {} {
    .label config -text [MaxLineLength [.e get]]
}
button .doit -text “Max Line Length” -command Doit
bind .e <Return> Doit
```

Chapter 13 describes the `bind` command in detail, and Chapter 15 describes the `label` widget, and Chapter 16 describes the `entry` widgets.

Buttons Associated with Tcl Variables

The `checkbutton` and `radiobutton` widgets are associated with a Tcl variable. When one of these buttons is clicked, a value is assigned to the Tcl variable. In addition, if the variable is assigned a value elsewhere in the program, the appearance of the check or radio button is updated to reflect the new value. A set of `radiobuttons` all share the same variable. The set represents a choice among mutually exclusive options. In contrast, each `checkbutton` has its own variable.

The `ShowChoices` example uses a set of radiobuttons to display a set of mutually exclusive choices in a user interface. The `ShowBooleans` example uses checkbuttons.

Example 14–4 Radio and Check buttons.


```
proc ShowChoices { parent varname args } {
    set f [frame $parent.choices -borderwidth 5]
    set b 0
    foreach item $args {
        radiobutton $f.$b -variable $varname \
            -text $item -value $item
        pack $f.$b -side left
        incr b
    }
    pack $f -side top
}
proc ShowBooleans { parent args } {
    set f [frame $parent.choices -borderwidth 5]
    set b 0
    foreach item $args {
        checkbutton $f.$b -text $item -variable $item
        pack $f.$b -side left
        incr b
    }
    pack $f -side top
}
set choice kiwi
ShowChoices {} choice apple orange peach kiwi strawberry
set Bold 1 ; set Italic 1
ShowBooleans {} Bold Italic Underline
```

The `ShowChoices` procedure takes as arguments the parent frame, the name of a variable, and a set of possible values for that variable. If the parent frame is null, `{}`, then the interface is packed into the main window . `ShowChoices` creates a `radiobutton` for each value, and it puts the value into the text of the button. It also has to specify the value to assign to the variable when the button is clicked. The default value is the name of the button, which would be the value of `b` in the example. Another way to define the radiobuttons and get the correct value would be like this:

```
radiobutton $f.$item -variable $varname -text $item
```

The danger of using `$item` as the button name is that not all values are legal widget names. If the value contained a period or began with a capital letter , the `radiobutton` command would raise an error . Tk uses periods, of course, to reflect the widget hierarchy in names. Capitalized names are reserved for X resource class names, so widget instance names cannot have capitalized components. Chapter 27 describes X resources in more detail.

The `ShowBooleans` procedure is similar to `ShowChoices`. It takes a set of variable names as arguments, and it creates a `checkbutton` for each variable. The default values for the variable associated with a `checkbutton` are zero and one, which is fine for this example. If you need particular values you can specify them with the `-onvalue` and `-offvalue` attributes.

Radio and check buttons can have commands associated with them, just like ordinary buttons. The command is invoked after the associated Tcl variable has been updated. Remember that the Tcl variable is modified in the global scope, so you need to access it with a global command if you use a procedure for your button command. For example, you could log the changes to variables as shown in the next example.

Example 14–5 Acommand on a `radiobutton` or `checkbutton`.

```
proc PrintByName { varname } {
    upvar #0 $varname var
    puts stdout "$varname = $var"
}
checkbutton $f.$b -text $item -variable $item \
    -command [list PrintByName $item]
radiobutton $f.$b -variable $varname \
    -text $item -value $item \
    -command [list PrintByName $varname]
```

Button Attributes

The table below lists the attributes for the `button`, `checkbutton`, `menubutton`, and `radiobutton` widgets. Unless otherwise indicated, the attributes apply to all of these widget types. Chapters 22, 23, and 24 discuss many of these attributes in more detail.

The table uses the X resource name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase. Compare:

```
option add *Menubutton.highlightColor: red
$mb configure -highlightcolor red
```

The first command defines a resource database entry that covers all menubutton widgets and gives them a red highlight. This only affects menubuttons created after the database entry is added. The second command changes an existing button (`.mb`) to have a red highlight. Note the difference in capitalization of `color` in the two commands. Chapter 27 explains how to use resource specifications for attributes.

Table 14–1 Resource names of attributes for all `button` widgets.

`activeBackground`	Background color when the mouse is over the button.
`activeForeground`	Text color when the mouse is over the button.
`anchor`	Anchor point for positioning the text.
`background`	The normal background color.
`bitmap`	A bitmap to display instead of text.
`borderWidth`	Width of the border around the button.
`command`	Tcl command to invoke when button is clicked.
`cursor`	Cursor to display when mouse is over the widget.
`disabledForeground`	Foreground (text) color when button is disabled.
`font`	Font for the text.
`foreground`	Foreground (text) color. (Also `fg`).
`height`	Height, in lines for text, or screen units for images.
`highlightColor`	Color for input focus highlight border.
`highlightThickness`	Width of highlight border.
`image`	Image to display instead of text or bitmap.
`indicatorOn`	Boolean that controls if the indicator is displayed. `checkbutton menubutton radiobutton`
`justify`	Text justification: `center left right`
`menu`	Menu posted when `menubutton` is clicked.
`offValue`	Value for Tcl variable when `checkbutton` is not selected.
`onValue`	Value for Tcl variable when `checkbutton` is selected.
`padX`	Extra space to the left and right of the button text.
`padY`	Extra space above and below the button text.
`relief`	3D relief: `flat, sunken, raised, groove, ridge`.
`selectColor`	Color for selector. `checkbutton radiobutton`
`selectImage`	Alternate graphic image for selector. `checkbutton radiobutton`
`state`	Enabled (`normal`) or deactivated (`disabled`).
`text`	Text to display in the button.
`textVariable`	Tcl variable that has the value of the text.
`underline`	Index of text character to underline.
`value`	Value for Tcl variable when `radiobutton` is selected.

Table 14–1 Resource names of attributes for all `button` widgets.

`variable`	Tcl variable associated with the button. `checkbutton radiobutton`
`width`	Width, in characters for text, or screen units for image.
`wrapLength`	Max character length before text is wrapped.

Button Operations

The table below summarizes the operations on button widgets. In the table `$w` is a `button`, `checkbutton`, `radiobutton`, or `menubutton`, For the most part these operations are used by the script libraries that implement the bindings for buttons. The `cget` and `configure` operations are the most commonly used.

Table 14–2 Button operations. .

$w cget *option*	Return the value of the specified attribute.
$w configure ?option? ?value" ...	Query or manipulate the configuration information for the widget.
$w flash	Redisplay the button several times in alternate colors.
$w invoke	Invoke the action associated with the button, just as if the user had pressed the mouse on it.

Menus and Menubuttons

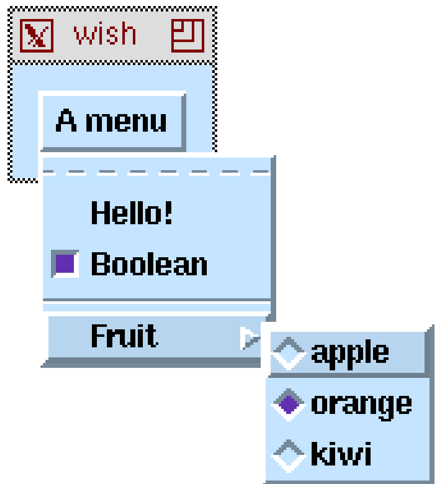
A `menu` presents a set of button-like *menu entries* to users. A menu entry is not a full fledged Tk widget. Instead, you create a menu widget and then add entries to the menu as shown below . There are several kinds of menu entries, including command entries, check entries, and radio entries. These all behave much like buttons, checkbuttons, and radiobuttons. Separator entries are used to visually set apart entries. Cascade entries are used to post sub-menus. T ear -off entries are used to detach a menu from its menu button so that it becomes a new top-level window .

A `menubutton` is a special kind of button that posts (i.e., displays) a menu when you press it. If you click on a menubutton, then the menu is posted and remains posted until you click on a menu entry to select it, or click outside the menu to dismiss it. If you press and hold the menubutton, then the menu is unposted when you release the mouse. If you release the mouse over the menu it selects the menu entry that was under the mouse.

You can have a command associated with a menubutton, too. The command is invoked *before* the menu is posted, which means you can compute the menu contents when the user presses the menubutton.

Our first menu example creates a sampler of the different entry types.

Example 14–6 A `menu` sampler.




```
menubutton .mb -text Sampler -menu .mb.menu
pack .mb -padx 10 -pady 10
set m [menu .mb.menu -tearoff 1]
$m add command -label Hello! -command {puts "Hello, World!"}
$m add check -label Boolean -variable foo \
   -command {puts "foo = $foo"}
$m add separator
$m add cascade -label Fruit -menu $m.sub1
set m2 [menu $m.sub1 -tearoff 0]
$m2 add radio -label apple -variable fruit
$m2 add radio -label orange -variable fruit
$m2 add radio -label kiwi -variable fruit
```

The example creates a menubutton and two menus. The main menu (`.mb.menu`) is a child of the menubutton (`.mb`). This relationship is necessary so the menu displays correctly when the menubutton is selected. Similarly , the cascaded submenu (`.mb.menu.sub1`) is a child of the main menu. The first menu entry is represented by the dashed line. This is a tear -off entry that, when selected, makes a copy of the menu in a new toplevel window . This is useful if the menu operations are invoked frequently . The `-tearoff 0` argument is used when creating the submenu to eliminate its tear -off entry .

The command, radio, and check entries are similar to the corresponding button types. The main difference is that the text string in the menu entry is defined the `-label` argument, not `-text`. The arguments to define the commands and variables associated with the menu entries are the same as for the button commands. T able 14–6 gives the complete set of attributes for menu entries.

The cascade menu entry is associated with another menu. It is distinguished by the small right arrow in the entry . When you select the entry the submenu is posted. It is possible to have several levels of cascaded menus. There is no hard limit to the number of levels, except that your users will complain if you nest menus too much.

Manipulating Menus and Menu Entries

A menu entry is referred to by an *index*. The index can be numerical, counting from 0. There are also some keyword indices, which are summarized in Table 14–3. One of the most useful indices is a pattern that matches the label in the menu entry. This form eliminates the need to keep track of the numerical indices.

Table 14–3 Menu entry index keywords

`index`	A numerical index counting from zero.
`active`	The activated entry, either because it is under the mouse or has been activated by keyboard traversal
`last`	The last menu entry.
`none`	No entry at all.
`@`*`ycoord`*	The entry under the given Y coordinate. Use `@%y` in bindings.
`pattern`	A string match pattern to match the label of a menu entry.

There are a number of operations that apply to menu entries. The `add` operation has been introduced already. The `entryconfigure` operation is similar to the configure operation. It accepts the same attribute-value pairs used when the menu entry was created. The `delete` operation removes a range of menu entries. The rest of the operations are used by the library scripts that implement the standard bindings for menus. The complete set of menu operations are summarized in the next table. In the table, `$w` is a menu widget..

Table 14–4 Menu operations.

`$w activate index`	Highlight the specified entry.
`$w add` *`type ?option value?`* `...`	Add a new menu entry of the specified type with the given values for various attributes.
`$w cget` *`option`*	Return the value for the configuration *`option`*.
`$w configure` *`?option? ?value?`* `...`	Return the configuration information for the menu.
`$w delete` *`i1 ?i2?`*	Delete the menu entries from index *`i1`* to *`i2`*
`$w entrycget` *`index option`*	Return the value of *`option`* for the specified menu entry.

Table 14–4 Menu operations.

`$w entryconfigure` *`index`* *`?option? ?value? ...`*	Query or modify the configuration information for the specified menu entry .
`$w index` *`index`*	Return the numerical index corresponding to *`index`*.
`$w invoke` *`index`*	Invoke the command associated with the entry .
`$w post` *`x y`*	Display the menu at the specified coordinates.
`$w type` *`index`*	Return the type of the entry at *index* .
`$w unpost`	Unmap the menu.
`$w ypostion` *`index`*	Return the y coordinate of the top of the menu entry .

A Menu by Name Package

If your application supports extensible or user -defined menus, then it can be tedious to expose all the details of the Tk menus. The examples in this section create a little package that lets users refer to menus and entries by name. In addition, the package supports keystroke accelerators for menus.

Example 14–7 A simple menu-by-name package.

```
proc MenuSetup { menubar } {
    global Menu
    frame $menubar
    pack $menubar -side top -fill x
    set Menu(menubar) $menubar
    set Menu(uid) 0
}
proc Menu { label } {
    global Menu
    if [info exists Menu(menu,$label)] {
        error "Menu $label already defined"
    }
    # Create the menubutton and its menu
    set name $Menu(menubar).mb$Menu(uid)
    set menuName $name.menu
    incr Menu(uid)
    set mb [menubutton $name -text $label -menu $menuName]
    pack $mb -side left
    set menu [menu $menuName -tearoff 1]
    # Remember the name to menu mapping
    set Menu(menu,$label) $menu
}
```

The `MenuSetup` procedure initializes the package. It creates a frame to hold the set of menu buttons, and it initializes some state variables: the frame for the menubuttons and a counter used to generate widget pathnames. All the global state for the package will be kept in the array called `Menu`.

The `Menu` procedure creates a `menubutton` and a `menu`. It records the association between the text label of the `menubutton` and the `menu` that was created for it. This mapping is used throughout the rest of the package so that the client of the package can refer to the menu by its label (e.g., `File`) as opposed to the internal Tk pathname, (e.g., `.top.menubar.file.menu`).

Example 14–8 Adding menu entries.

```
proc MenuCommand { menuName label command } {
	global Menu
	if [catch {set Menu(menu,$menuName)} menu] {
		error "No such menu: $menuName"
	}
	$menu add command -label $label -command $command
}

proc MenuCheck { menuName label var { command {} } } {
	global Menu
	if [catch {set Menu(menu,$menuName)} menu] {
		error "No such menu: $menuName"
	}
	$menu add check -label $label -command $command \
		-variable $var
}

proc MenuRadio { menuName label var {val {}} {command {}} } {
	global Menu
	if [catch {set Menu(menu,$menuName)} menu] {
		error "No such menu: $menuName"
	}
	if {[string length $val] == 0} {
		set val $label
	}
	$menu add radio -label $label -command $command \
		-value $val -variable $var
}

proc MenuSeparator { menuName } {
	global Menu
	if [catch {set Menu(menu,$menuName)} menu] {
		error "No such menu: $menuName"
	}
	$menu add separator
}
```

The procedures `MenuCommand`, `MenuCheck`, `MenuRadio`, and `MenuSeparator` are simple wrappers around the basic menu commands. The only trick is that they use the `Menu` variable to map from the menu label to the Tk widget name. If the user specifies a bogus menu name, the undefined variable error is caught and a more informative error is raised instead.

Creating a cascaded menu also requires saving the mapping between the label in the cascade entry and the Tk pathname for the submenu. This package imposes a restriction that different menus, including submenus cannot have the same label.

Example 14–9 A wrapper for cascade entries.

```
proc MenuCascade { menuName label } {
    global Menu
    if [catch {set Menu(menu,$menuName)} menu] {
        error "No such menu: $menuName"
    }
    if [info exists Menu(menu,$label)] {
        error "Menu $label already defined"
    }
    set sub $menu.sub$Menu(uid)
    incr Menu(uid)
    menu $sub -tearoff 0
    $menu add cascade -label $label -menu $sub
    set Menu(menu,$label) $sub
}
```

Creating the sampler menu with this package looks like this:

Example 14–10 Using the basic menu package.

```
MenuSetup
Menu Sampler
MenuCommand Sampler Hello! {puts "Hello, World!"}
MenuCheck Sampler Boolean foo {puts "foo = $foo"}
MenuSeparator Sampler
MenuCascade Sampler Fruit
MenuRadio Fruit apple fruit
MenuRadio Fruit orange fruit
MenuRadio Fruit kiwi fruit
```

The final touch on the menu package is to support accelerators in a consistent way . A menu entry can display another column of information that is assumed to be a keystroke identifier to remind users of a binding that also invokes the menu entry . However , there is no guarantee that this string is correct, or that if the user changes the binding that the menu will be updated. The `MenuBind` procedure takes care of this.

Example 14–11 Keeping the accelerator display up-to-date.

```
proc MenuBind { what sequence menuName label } {
    global Menu
    if [catch {set Menu(menu,$menuName)} menu] {
        error "No such menu: $menuName"
    }
```

```
    if [catch {$menu index $label} index] {
        error "$label not in menu $menuName"
    }
    set command [$menu entrycget $index -command]
    bind $what $sequence $command
    $menu entryconfigure $index -accelerator $sequence
}
```

The `MenuBind` command uses the `index` operation to find out what menu entry has the given label. It updates the display of the entry using the `entryconfigure` operation, and it creates a binding using the `bind` command. This approach has the advantage of keeping the keystroke command consistent with the menu command, as well as updating the display. To try out `MenuBind`, add an empty frame to the sampler example, and bind a keystroke to it and one of the menu commands, like this:

```
frame .body -width 100 -height 50
pack .body ; focus .body
MenuBind .body <space> Sampler Hello!
```

Popup Menus and Option Menus

The Tk script library comes with two procedures that are used to create *popup* menus and *option* menus. A popup menu is not associated with a button. Instead, it is posted in response to a keystroke or other event in the application. An option menu represents a choice with a set of radio entries, and it displays the current choice in the text of the menu button.

The `tk_popup` command posts a popup menu. First, create the menu as described above, except that you do not need a menubutton. Then post the popup menu like this:

```
tk_popup $menu $x $y $entry
```

The last argument specifies the entry to activate when the menu is posted. It is an optional parameter that defaults to 1. The menu is posted at the specified X and Y coordinates in its parent widget.

The `tk_optionMenu` command creates a menubutton and a menu full of radio entries. It is invoked like this:

```
tk_optionMenu w varname firstValue ?value value ...?
```

The first argument is the pathname of the menubutton to create. The second is the variable name. The third is the initial value for the variable, and the rest or the other choices for the value. The menubutton displays the current choice and a small symbol, the indicator, to indicate it is a choice menu.

Keyboard Traversal

The default bindings for menus allow for keyboard selection of menu entries. The selection process is started by pressing `<Alt-`*x*`>` where *x* is the distinguishing letter for a menubutton. The `underline` attribute of a `menubutton` is used to highlight the appropriate letter. The underline value is a number that specifies a character position, and the count starts at zero. For example, a `File` menu with a highlighted `F` is created like this:

```
menubutton .menubar.file -text File -underline 0 \
    -menu .menubar.file.m
```

When the user types `<Alt-f>` over the main window, the menu `.menubar.-file.m` is posted. The case of the letter is not important.

After a menu is posted the arrow keys can be used to change the selected entry. The `<Up>` and `<Down>` keys move within a menu, and the `<Left>` and `<Right>` keys move between adjacent menus. The bindings assume that you create your menus from left to right.

If any of the menu entries have a letter highlighted with the `-underline` option, then typing that letter will invoke that menu entry. For example, an `Export` entry that is invoked by typing `x` can be created like this:

```
.menubar.file.m add command -label Export -underline 1 \
    -command File_Export
```

The `<space>` and `<Return>` keys will invoke the menu entry that is currently selected. The `<Escape>` key will abort the menu selection and unpost the menu.

Menu Attributes

A menu has a few global attributes, and then each menu entry has many button-like attributes that describe its appearance and behavior. The table below gives the attributes that apply globally to the menu, unless overridden by a per-entry attribute. The table uses the X resource names, which may have a capital at interior word boundaries. In Tcl commands use all lowercase and a leading dash.

Table 14–5 Resource names of attributes for `menu` widgets.

`activeBackground`	Background color when the mouse is over a menu entry.
`activeForeground`	Text color when the mouse is over a menu entry.
`activeBorderWidth`	Width of the raised border around active entries.
`background`	The normal background color for menu entries.
`borderWidth`	Width of the border around all the menu entries.
`cursor`	Cursor to display when mouse is over the menu.
`disabledForeground`	Foreground (text) color when menu entries are disabled.
`font`	Default font for the text.
`foreground`	Foreground color. (Also `fg`).

Table 14–5 Resource names of attributes for `menu` widgets.

`postCommand`	Tcl command to run just before menu is posted.
`selectColor`	Color for selector in check and radio type entries.
`tearOff`	True if menu should contain a tear off entry.

The attributes for menu entries are only valid in Tcl commands; they are not supported directly by the X resource database. You can still use the resource database for menu entries as described in Example 27–5 on page 328. The table below describes the attributes for menu entries, as you would use them in a Tcl command (i.e., all lowercase with a leading dash.).

Table 14–6 Attributes for menu entries.

`-activebackground`	Background color when the mouse is over the entry.
`-activeforeground`	Foreground (text) color with mouse is over the entry.
`-accelerator`	Text to display as a reminder about keystroke binding.
`-background`	The normal background color.
`-bitmap`	A bitmap to display instead of text.
`-command`	Tcl command to invoke when entry is invoked.
`-font`	Default font for the text.
`-foreground`	Foreground color. (Also `fg`).
`-image`	Image to display instead of text or bitmap.
`-label`	Text to display in the menu entry.
`-justify`	Text justification: `center left right`
`-menu`	Menu posted when `cascade` entry is invoked.
`-offvalue`	Value for Tcl variable when `checkbutton` entry is not selected.
`-onvalue`	Value for Tcl variable when `checkbutton` entry is selected.
`-selectcolor`	Color for selector. `checkbutton and radiobutton` entries.
`-state`	`normal active disabled`
`-underline`	Index of text character to underline.
`-value`	Value for Tcl variable when `radiobutton` entry is selected.
`-variable`	Tcl variable associated with the `checkbutton` or `radiobutton` entry.

CHAPTER 15

Simple Tk Widgets

This chapter describes several simple Tk widgets: the frame, label, message, scale, and scrollbar. In general, these widgets require minimal setup to be useful in your application. The bell command rings the X display bell, and doesn't fit into other chapters, so it is described here.

This chapter describes five simple widgets. The `frame` is a building block for widget layout. The `label` provides a line of read-only text. The `message` provides a read-only block of text that gets formatted onto several lines. The `scale` is a slider-like widget used to set a numeric value. The `scrollbar` is used to control other widgets. These widgets (and the `bell` command) are not that interesting by themselves, so this chapter reviews their functions rather briefly.

Chapter 22, 23, and 24 go into more detail about some of the generic widget attributes shared by the widgets presented in this chapter. The examples in this chapter use the default widget attributes in most cases.

Frames and Top-Level Windows

Frames have been introduced before for use with the geometry managers. There is not much to a `frame`, except for its background color and border. You can also specify a colormap and visual type for a frame. Chapter 23 describes visual types and colormaps in more detail.

A `toplevel` widget is like a frame, except that it is created as a new toplevel window. That is, it is not positioned inside the main window of the application. This is useful for dialog boxes, for example. A `toplevel` has the same attributes as a frame, plus it has a screen option that lets you put the toplevel on any X display . The value of the screen option uses the same format that you use when you start an application

host:*display*.*screen*

For example, I have one X server on my workstation `corvina` that controls two screens. My two screens are named `corvina:0.0` and `corvina:0.1`. If the screen specifier is left off, it defaults to `0`.

Attributes for frames and toplevels

Table 15–2 lists the attributes for the `frame` and `toplevel` widgets. The attributes are named according to their X resource name, which includes a capital letter at internal word boundaries. When you specify an attribute in a T cl command when creating or reconfiguring a widget, however , you specify the attribute with a dash and all lowercase letters. Chapter 27 explains how to use resource specifications for attributes . Chapters 22, 23, and 24 discuss many of these attributes in more detail.

Table 15–1 Resource names of attributes for `frame` and `toplevel` widgets.

`background`	Background color (also `bg`).
`borderWidth`	Extra space around the edge of the label.
`class`	X resource class and binding class name.
`colormap`	The value is `new` or the name of a window .
`cursor`	Cursor to display when mouse is over the label.
`height`	In screen units for bitmaps, in lines for text.
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`screen`	An X display specification. (`toplevel` only, and this cannot be specified in the resource database.)
`visual`	`staticgrey greyscale staticcolor pseudocolor directcolor truecolor`
`width`	Width. In characters for text labels.

The `class`, `colormap`, `visual`, and `screen` attributes cannot be changed after the `frame` or `toplevel` has been created. These settings are so fundamental that you basically need to destroy the frame and start over if you have to change one of these.

The label Widget

The `label` widget provides a read-only text label, plus it has attributes that let you control the position of the label within the display space. Most commonly, however, you just need to specify the text for the label.

```
label .version -text "MyApp v1.0"
```

The text can be specified indirectly by using a Tcl variable to hold the text. In this case the label will be updated whenever the value of the Tcl variable changes. The variable is used from the global scope, even if there happens to be a local variable by the same name when you create the widget inside a procedure.

```
set version "MyApp v1.0"
label .version -textvariable version
```

The appearance of a label can be changed dynamically by using the `configure` widget operation. If you change the text or font of a label you are liable to change the size of the widget, and this will cause the packer to shuffle window positions. You can avoid this by specifying a width for the label that is large enough to hold all the strings you plan to display in it. The width is specified in characters, not screen coordinates.

Example 15–1 A `label` that displays different strings.

```
proc FixedWidthLabel { name values } {
    # name is a widget name to be created
    # values is a list of strings
    set maxWidth 0
    foreach value $values {
        if {[string length $value] > $maxWidth} {
            set maxWidth [string length $value]
        }
    }
    # Use -anchor w to left-justify short strings
    label $name -width $maxWidth -anchor w \
        -text [lindex $values 0]
    return $name
}
```

The `FixedWidthLabel` example is used to create a label with a width big enough to hold a set of different strings. It uses the `-anchor w` attribute to left justify strings that are shorter than the maximum. The text for the label can be changed later by using the configure widget command:

```
FixedWidthLabel .status {OK Busy Error}
.status config -text Busy
```

A label can display a bitmap instead of a text string. For a discussion of using bitmaps, see Chapter 23 and the section on *Bitmaps and Images*.

Label attributes

Table 15–2 lists the widget attributes for the `label` widget. The attributes are named according to their X resource name, which includes a capital letter at internal word boundaries. When you specify an attribute in a Tcl command when creating or reconfiguring a widget, however, you specify the attribute with a dash and all lowercase letters. Chapter 27 explains how to use resource specifications for attributes. Chapters 22, 23, and 24 discuss many of these attributes in more detail.

Table 15–2 Resource names of attributes for `label` widgets.

`anchor`	Relative position of the label within its packing space.
`background`	Background color (also `bg`).
`bitmap`	Name of a bitmap to display instead of a text string.
`borderWidth`	Extra space around the edge of the label.
`cursor`	Cursor to display when mouse is over the label.
`font`	Font for the label's text.
`foreground`	Foreground color. (Also `fg`).
`height`	In screen units for bitmaps, in lines for text.
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`image`	Specifies image to display instead of bitmap or text.
`justify`	Text justification: `left`, `right`, `center`.
`padX`	Extra space to the left and right of the label.
`padY`	Extra space above and below the label.
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`text`	Text to display.
`textVariable`	Name of Tcl variable. Its value is displayed.
`underline`	Index of character to underline.
`width`	Width. In characters for text labels.
`wrapLength`	Length at which text is wrapped *in screen units*.

Label width and wrapLength

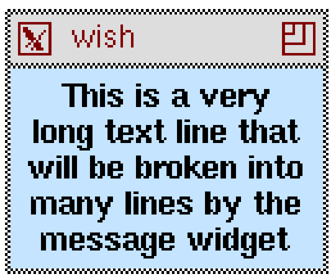
When a label is displaying text, its `width` attribute is interpreted as a number of characters. The label is made wide enough to hold this number of averaged width characters in the label's font. However, if the label is holding a bitmap or an image, then the `width` is in pixels or another screen unit.

The `wrapLength` attribute determines when a label's text is wrapped onto multiple lines. *The wrap length is always screen units*. If you need to compute a `wrapLenth` based on the font metrics (instead of guessing) then you'll have to use a `text` widget with the same font. Chapter 18 describes the `text` widget operations that return size information for characters.

The message Widget

The `message` widget displays a long text string by formatting it onto several lines. It is designed for use in dialog boxes. It can format the text into a box of a given width, in screen units, or a given *aspect ratio*. The aspect ratio is defined to be the ratio of the width to the height, times 100. The default is 150, which means the text will be one and a half times as wide as it is high.

Example 15–2 The `message` widget formats long lines of text.

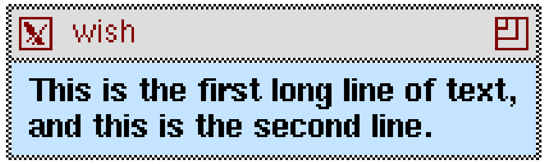



```
message .msg -justify center -text "This is a very long text\
   line that will be broken into many lines by the\
   message widget"
pack .msg
```

This example creates a message widget with one long line of text. Backslashes are used to continue the text string without embedding any newlines. (You can also just type a long line into your script.) Note that backslash-newline collapses white space after the newline into a single space.

A newline in the string forces a line break in the message display. You can retain exact control over the formatting by putting newlines into your string and specifying a very large aspect ratio. In the next example, grouping with double quotes is used to continue the string over more than one line. The newline character between the quotes is included in the string, and it causes a line break.

Example 15–3 Controlling the text layout in a message widget.




```
message .msg -aspect 1000 -justify left -text \
"This is the first long line of text,
and this is the second line."
pack .msg
```

Message Attributes

The table on the next page lists the attributes for the message widget. The table list the X resource name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 15–3 Resource names for attributes for `message` widgets.

`anchor`	Relative position of the text within its packing space.
`aspect`	100 * width / height. Default 150.
`background`	Background color (also `bg`).
`borderWidth`	Extra space around the edge of the text.
`cursor`	Cursor to display when mouse is over the widget.
`font`	Font for the label's text.
`foreground`	Foreground color. (Also `fg`).
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`justify`	`left`, `center`, or `right`. Defaults to `left`.
`padX`	Extra space to the left and right of the text.
`padY`	Extra space above and below the text.
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`text`	Text to display.
`textVariable`	Name of Tcl variable. Its value is displayed.
`width`	Width, in screen units.

Arranging Labels and Messages

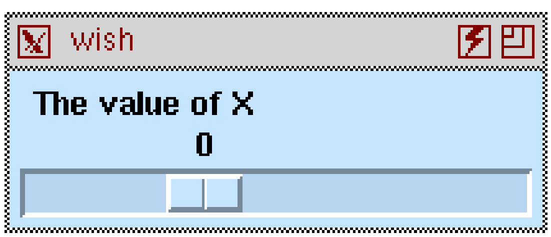
Both the `label` and `message` widgets have attributes that control the position of their text in much the same way that the packer controls the position of widgets within a frame. These attributes are `padX`, `padY`, `anchor` and `borderWidth`. The `anchor` takes effect when the size of the widget is larger than the space needed to display its text. This happens when you specify the `-width` attribute or if you pack the widget with filling enabled and there is extra room. See Chapter 22 and the section on *Padding and Anchors* for more details.

The scale Widget

The `scale` widget displays a *slider* in a *trough*. The trough represents a range of numeric values, and the slider position represents the current value. The `scale` can have an associated label, and it can display its current value next to the slider.

The value of the `scale` can be used in three different ways. You can explicitly `get` and `set` the value with widget commands. You can associate the `scale` with a Tcl variable. The variable is kept in sync with the value of the `scale`, and changing the variable affects the `scale`. Finally, you can arrange for a Tcl command to be executed when the `scale` value changes. You specify the initial part of the Tcl command, and the `scale` implementation adds the current value as another argument to the command.

Example 15–4 A `scale` widget.




```
scale .scale -from -10 -to 20 -length 200 -variable x \
    -orient horizontal -label "The value of X" \
    -command myprint
proc myprint { value } {puts "The value of X is $value"}
pack .scale
```

The example shows a `scale` that has both a variable and a command. Typically you would use just one of these options. The `myprint` procedure can get the value in two ways. As well as using its argument, it could use a `global x` command to make the scale variable visible in its scope.

The scale has a `resolution` and `bigIncrement` attribute that determine how its value can be changed. If the resolution is set to 0.1, for example, then the value will be rounded to the nearest tenth. The bigIncrement attribute is used in the keyboard bindings to shift the value by a large amount. Table 15–4 lists the bindings for scale widgets.

Table 15–4 Default bindings for scale widgets.

<Button-1>	Clicking on the trough moves the slider by one unit of resolution towards the mouse click.
<Control-Button-1>	Clicking on the trough moves the slider all the way to the end of the trough towards the mouse click.
<Left> <Up>	Move the slider towards the left (top) by one unit.
<Control-Left> <Control-Up>	Move the slider towards the left (top) by the value of the bigIncrement attribute.
<Right> <Down>	Move the slider towards the right (bottom) one unit.
<Control-Right> <Control-Down>	Move the slider towards the right (bottom) by the value of the bigIncrement attribute.
<Home>	Move the slider all the way to the left (top).
<End>	Move the slider all the way to the right (bottom).

Scale attributes

The following table lists the scale widget attributes. The table uses the X resource Class name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 15–5 Resource names for attributes for scale widgets.

activeBackground	Background color when the mouse is over the slider.
background	The background color. (Also bg in commands.)
bigIncrement	Coarse grain slider adjustment value.
borderWidth	Extra space around the edge of the text.
command	Command to invoke when the value changes. The current value is appended as another argument
cursor	Cursor to display when mouse is over the widget.
digits	Number of significant digits in scale value.
from	Minimum value. The left or top end of the scale.
font	Font for the label.
foreground	Foreground color. (Also fg).
highlightColor	Color for input focus highlight.

Table 15–5 Resource names for attributes for `scale` widgets.

`highlightThickness`	Thickness of focus highlight rectangle.
`label`	A string to display with the scale.
`length`	The length, in screen units, of the long axis of the scale.
`orient`	`horizontal` or `vertical`
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`repeatDelay`	Delay before keyboard auto-repeat starts. Auto-repeat is used when pressing `<Button-1>` on the trough.
`repeatInterval`	Time period between auto-repeat events.
`resolution`	The value is rounded to a multiple of this value.
`showValue`	If true, value is displayed next to the slider.
`sliderLength`	The length, in screen units, of the slider.
`state`	`normal`, `active`, or `disabled`
`tickInterval`	Spacing between tick marks. Zero means no marks.
`to`	Maximum value. Right or bottom end of the scale.
`troughColor`	The color of the bar on which the slider sits.
`variable`	Name of Tcl variable. Changes to the `scale` widget are reflected in the Tcl variable value, and changes in the Tcl variable are reflected in the `scale` display.
`width`	Width of the trough, or slider bar.

Programming scales

The `scale` widget supports a number of operations. For the most part these are used by the default bindings and you won't need to program the `scale` directly. Table 15–6 lists the operations supported by the scale. In the table, `$w` is a `scale` widget.

Table 15–6 Operations on `scale` widgets..

`$w cget` *`option`*	Return the value of the configuration option.
`$w configure ...`	Query or modify the widget configuration.
`$w coords ?`*`value`*`?`	Returns the coordinates of the point in the trough that corresponds to *`value`*, or the scale's value.
`$w get ?x y?`	Return the value of the scale, or the value that corresponds to the position given by *`x`* and *`y`*.
`$w identify` *`x y`*	Returns `trough1`, `slider`, or `trough2` to indicate what is under the position given by *`x`* and *`y`*

Table 15–6 Operations on `scale` widgets..

`$w set` *`value`*	Set the value of the scale.

The scrollbar Widget

The `scrollbar` is used to control the display of another widget. The Tk widgets designed to work with scrollbars are the `entry`, `listbox`, `text`, and `canvas` widgets. There is a simple protocol between the `scrollbar` and these widgets. While this section explains the protocol, you don't need to know the details to use a scrollbar. All you need to know is how to set things up, and then these widgets take care of themselves.

A scrollbar is made up of 5 components: `arrow1`, `trough1`, `slider`, `trough2`, and `arrow2`. The arrows are on either end, with `arrow1` being the arrow to the left for horizontal scrollbars, or the arrow on top for vertical scrollbars. The slider represents the relative position of the information displayed in the associated widget, and the size of the slider represents the relative amount of the information displayed. The two trough regions are the areas between the slider and the arrows. If the slider covers all of the trough area, you can see all the information in the associated widget.

The protocol between the `scrollbar` and its associated widget (or widgets) is initialized by registering a command with each of the widgets. The `scrollbar` has a `command` attribute that is uses to scroll the associated widget. The `xview` and `yview` operations of the scrollable widgets are designed for this. These operations require parameters that indicate how to adjust their view, and the scrollbar adds these parameters when it calls the command. The command to create a `scrollbar` for a `text` widget would look something like this:

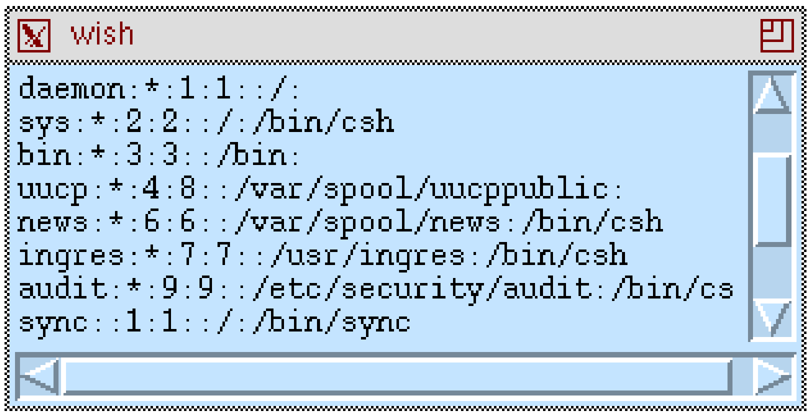
```
scrollbar .scroll -command {.text yview} -orient vertical
```

The scrollable widgets have `xscrollcommand` and/or `yscrollcommand` attributes that they use to update the display of the scrollbar. The `scrollbar set` operation is designed for this callback. Additional parameters are appended to these commands that indicate how much information is visible in the widget and the relative position of that information. The command below sets up the other half of the relationship between the `scrollbar` and the `text` widget.

```
text .text -yscrollcommand {.scroll set}
```

The protocol works like this. When the scrollbar is manipulated by the user it calls its registered `command` with some parameters that indicate what the user said to do. The associated widget responds to this command (e.g., its `xview` operation) by changing its display. After the widget changes its display, it calls the `scrollbar` by using its registered `xscrollcommand` or `yscrollcommand` (e.g., the `set` operation) with some parameters that indicate the new relative size and position of the display. The scrollbar updates its appearance to reflect this information. The protocol supports widgets that change their display by themselves, such as by dragging them with `<B2-Motion>` events or simply by adding more information. When this happens, the `scrollbar` will be updated correctly, even though it did not cause the display change.

Example 15–5 A `text` widget and two scrollbars.




```
proc ScrolledText { f width height } {
	frame $f
	# The setgrid setting allows the window to be resized.
	text $f.text -width $width -height $height \
		-setgrid true -wrap none \
		-xscrollcommand [list $f.xscroll set] \
		-yscrollcommand [list $f.yscroll set]
	scrollbar $f.xscroll -orient horizontal \
		-command [list $f.text xview]
	scrollbar $f.yscroll -orient vertical \
		-command [list $f.text yview]
	pack $f.xscroll -side bottom -fill x
	pack $f.yscroll -side right -fill y
	# The fill and expand are needed when resizing.
	pack $f.text -side left -fill both -expand true
	pack $f -side top -fill both -expand true
	return $f.text
}
set t [ScrolledText .f 40 8]
set in [open /etc/passwd]
$t insert end [read $in]
close $in
```

The example associates a `text` widget with two scrollbars. It reads and inserts the password file into the text widget. There is not enough room to display all the text, and the scrollbars indicate how much text is visible. Chapter 18 describes the `text` widget in more detail.

Table 15–4 lists the default bindings for scrollbars. Button 1 and button 2 of the mouse have the same bindings. A scrollbar does not normally get the keyboard focus, so you will have to direct the focus to it explicitly for the key bindings like <Up> and <Down> to take effect.

Table 15–7 Default bindings for `scrollbar` widgets.

`<Button-1> <Button-2>`	Clicking on the arrows scrolls by one unit. Clicking on the trough moves by one screenful.
`<B1-Motion> <B2-Motion>`	Dragging the slider scrolls dynamically.
`<Control-Button-1>` `<Control-Button-2>`	Clicking on the trough or arrow scrolls all the way to the beginning (end) of the widget.
`<Up> <Down>`	Scroll up (down) by one unit
`<Control-Up>` `<Control-Down>`	Scroll up (down) by one screenful.
`<Left> <Right>`	Scroll left (right) by one unit.
`<Control-Left>` `<Control-Right>`	Scroll left (right) by one screenful.
`<Prior> <Next>`	Scroll back (forward) by one screenful.
`<Home>`	Scroll all the way to the left (top).
`<End>`	Scroll all the way to the right (bottom).

Scrollbar attributes

Table 15–8 lists the scrollbar attributes. The table uses the X resource name for the attribute, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 15–8 Resource names of attributes for `scrollbar` widgets.

`activeBackground`	Color when the mouse is over the slider or arrows.
`activeRelief`	Relief of slider and arrows when mouse is over them.
`background`	The background color. (Also `bg` in commands.)
`borderWidth`	Extra space around the edge of the scrollbar.
`command`	Prefix of the command to invoke when the scrollbar changes. Typically this is a `xview` or `yview` operation.
`cursor`	Cursor to display when mouse is over the widget.
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`jump`	If true, dragging the elevator does not scroll dynamically. Instead, the display jumps to the new position.
`orient`	`horizontal` or `vertical`
`repeatDelay`	Delay before keyboard auto-repeat starts. Auto-repeat is used when pressing `<Button-1>` on the trough or arrows.

Table 15–8 Resource names of attributes for `scrollbar` widgets.

`repeatInterval`	Time period between auto-repeat events.
`troughColor`	The color of the bar on which the slider sits.
`width`	Width of the narrow dimension of the scrollbar.

There is no `length` attribute for a scrollbar. Instead, a `scrollbar` is designed to be packed next to another widget with a fill option that lets the `scrollbar` display grow to the right size. The relief of the `scrollbar` cannot be changed from `raised`. Only the relief of the active element can be set. The `background` color is used for the slider, the arrows, and the border The slider and arrows are displayed in the `activeBackground` color when the mouse is over them. The trough is always displayed in the `troughColor`.

Programming scrollbars

The `scrollbar` widget supports a number of operations. However, for the most part these are used by the default bindings. Table 15–6 lists the operations supported by the `scrollbar`. In the table, `$w` is a `scrollbar` widget.

Table 15–9 Operations on scrollbar widgets.

`$w activate` *`?element?`*	Query or set the active element, which can be `arrow1`, `arrow2`, or `slider`.
`$w cget` *`option`*	Return the value of the configuration option.
`$w configure ...`	Query or modify the widget configuration.
`$w fraction` *`x y`*	Return a number between 0 and 1 that indicates the relative location of the point in the trough.
`$s get`	Return *`first`* and *`last`* from the set operation.
`$w identify` *`x y`*	Returns `arrow1`, `trough1`, `slider`, `trough2`, or arrow2, to indicate what is under the point.
`$w set` *`first last`*	Set the scrollbar parameters. *`first`* is the relative position of the top (left) of the display. *`last`* is the relative position of the bottom (right) of the display.

The Tk 3.6 protocol

The protocol between the `scrollbar` and its associated widget changed in Tk 4.0. The `scrollbar` is backward compatible. The old protocol had 4 parameters in the `set` operation: *`totalUnits`*, *`windowUnits`*, *`firstUnit`*, and *`lastUnit`*. If a `scrollbar` is updated with this form of a `set` command, then the `get` operation also changes to return this information. When the `scrollbar` makes the callback to the other widget (e.g., an `xview` or `yview` operation), it passes a single extra parameter that specifies what *unit* to display at the top (left) of the associated widget. The Tk widgets' `xview` and `yview` operations are also backward compatible with this interface.

The bell Command

The `bell` command rings the X display bell. About the only interesting property of the bell is that it is associated with the display, so even if you are executing your program on a remote machine, the bell is heard by the user. If your application has windows on multiple displays, you can direct the bell to the display of a particular window with the `-displayof` option. The syntax for the bell command is given below:

```
bell ?-displayof window?
```

If you want to control the bell's duration, pitch, or volume, you need to use the *xset* program. The volume is in percent of a maximum, e.g. 50. In practice, many keyboard bells only support a variable duration, and the pitch is fixed. The arguments of *xset* that controll the bell are shown below.

```
exec xset b ?volume? ?hertz? ?milliseconds?
```

The `b` argument by itself resets the bell to the default parameters. You can turn the bell off with `-b`, or you can use the `on` or `off` arguments.

```
exec xset -b
exec xset b ?on? ?off?
```

CHAPTER 16

Entry and Listbox Widgets

The entry widget provides a single line of text for use as a data entry field. The listbox provides a scrollable list of text lines.

Listbox and entry widgets are specialized text widgets. They provide a subset of the functionality of the general purpose text widget. They are a bit more complex than the simple widgets presented in the previous chapter. You are more likely to program behavior for these widgets, especially the listbox.

The entry Widget

The `entry` widget provides a one-line type-in area. It is commonly used in dialog boxes when values need to be filled in, or as a simple command entry widget. The entry widget supports editing, scrolling, and selections, which make it quite a bit more complex than label or message widgets. Fortunately, the default settings for an `entry` widget make it usable right away. You click with the left button to set the insert point, and then type in text. Text is selected by dragging out a selection with the left button. The entry can be scrolled horizontally by dragging with the middle mouse button.

The complete set of bindings is given in the table below. When the table lists two sequences they are equivalent. For example, both the left arrow key (`<Left>`) and `<Control-b>` move the insert cursor to the left by one character. The table does not list all the right arrow key bindings, although there are corresponding bindings for the left and right arrow keys. The middle mouse button

(`<Button-2>`) is overloaded with two functions. If you click and release the middle button, then the selection is inserted at the insert cursor. The location of the middle click does not matter. If you press and hold the middle button, then you can scroll the contents of the entry by dragging the mouse to the left or right.

Table 16–1 Default bindings for `entry` widgets.

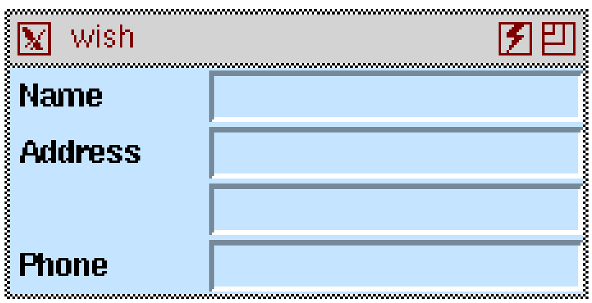
`<Button-1>`	Set insert point in start a selection.
`<B1-Motion>`	Drag out a selection.
`<Double-Button-1>`	Select a word.
`<Triple-Button-1>`	Select all text in the entry.
`<Shift-B1-Motion>`	Adjust the ends of the selection.
`<Control-Button-1>`	Set insert point, leaving selection as is.
`<Button-2>`	Paste selection at the insert cursor.
`<B2-Motion>`	Scroll horizontally.
`<Left> <Control-b>`	Move insert cursor one character left. Start selection.
`<Shift-Left>`	Move cursor left and extend selection.
`<Control-Left>`	Move cursor left one word. Start selection.
`<Meta-b>`	Same as <Control-Left>
`<Control-Shift-Left>`	Move cursor left one word and extend the selection.
`<Right> <Control-f>`	The bindings for Right correspond to the Left key.
`<Meta-f>`	Same as <Control-Right>, move right one word.
`<Home> <Control-a>`	Move cursor to beginning of entry.
`<Shift-Home>`	Move cursor to beginning and extend the selection.
`<End> <Control-e>`	Move cursor to end of entry.
`<Shift-End>`	Move cursor to end and extend the selection.
`<Select> <Control-Space>`	Anchor the selection at the insert cursor.
`<Shift-Select>` `<Control-Shift-Space>`	Adjust the selection to the insert cursor.
`<Control-slash>`	Selects all the text in the entry.
`<Control-backslash>`	Clears the selection in the entry.
`<Delete>`	Delete the selection or delete next character.
`<Backspace> <Control-h>`	Delete the selection or delete previous character.
`<Control-d>`	Delete next character.
`<Meta-d>`	Delete next word.

Table 16–1 Default bindings for entry widgets.

<Control-k>	Delete to the end of the entry.
<Control-w>	Delete previous word.
<Control-x>	Delete the section, if it exists.
<Control-t>	Transpose characters.

One common use of an entry widget is to associate a label with it, and a command to execute when <Return> is pressed in the entry. This is implemented in the following example.

Example 16–1 A command, a label and an entry.




```
proc CommandEntry { name label width command args } {
    frame $name
    label $name.label -text $label -width $width -anchor w
    eval {entry $name.entry -relief sunken} $args
    pack $name.label -side left
    pack $name.entry -side right -fill x -expand true
    bind $name.entry <Return> $command
    return $name.entry
}
CommandEntry .name Name 10 UpdateAddress -textvar addr(name)
CommandEntry .address1 Address 10 UpdateAddress \
    -textvar addr(line1)
CommandEntry .address2 "" 10 UpdateAddress \
    -textvar addr(line2)
CommandEntry .phone Phone 10 UpdateAddress \
    -textvar addr(phone)
pack .name .address1 .address2 .phone
```

CommandEntry creates a frame to hold the label and the entry widget. The label and lwidth arguments are used to define the label. The explicit width and the -anchor w are used so that you can line up the labels if you have more than one CommandEntry. The label is packed first so it does not get clipped if the frame is made too small. The entry is packed so it will fill up any extra space, if any. The args parameter is used to pass extra parameters along to the entry widget.

This requires the use of `eval` as discussed in Chapter 6. The Tcl command is bound to the `<Return>` keystroke. Finally, the pathname of the entry widget is returned in case the caller needs it.

The example includes four sample calls to `CommandEntry` and the `pack` command used to arrange them. The `-relief sunken` for the `entry` widget sets them apart visually, and you can see the effect of the `-anchor w` on the labels. The `-textvar` attribute is used to associate a Tcl variable with the entries, and in this case array elements are specified. The Tcl command `UpdateAddress` can get the current values of the entry widgets through the global array variable `addr`.

entry attributes

The following table lists the `entry` widget attributes. The table lists the X resource name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 16–2 Resource names for attributes of `entry` widgets.

`background`	Background color (also `bg`).
`borderWidth`	Extra space around the edge of the text (also `bd`).
`cursor`	Cursor to display when mouse is over the widget.
`exportSelection`	If "true", then the selected text is exported via the X selection mechanism.
`font`	Font for the text.
`foreground`	Foreground color. (Also `fg`).
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`insertBackground`	Background for area covered by insert cursor.
`insertBorderWidth`	Width of cursor border. Non-zero for 3D effect.
`insertOffTime`	Time, in milliseconds the insert cursor blinks off.
`insertOnTime`	Time, in milliseconds the insert cursor blinks on.
`insertWidth`	Width of insert cursor. Default is 2.
`justify`	Text justification: `left`, `right`, `center`.
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`selectBackground`	Background color of selection.
`selectForeground`	Foreground color of selection.
`selectBorderWidth`	Widget of selection border. Non-zero for 3D effect.
`show`	If false, asterisk (*) are displayed instead of contents.

Table 16–2 Resource names for attributes of `entry` widgets.

`state`	`disabled` (read-only) or `normal`.
`textVariable`	Name of Tcl variable.
`width`	Width, in characters.
`xScrollCommand`	Used to connect entry to a scrollbar.

Perhaps the most useful attribute of an entry widget is the `textVariable` attribute. Use this to mirror the contents of the `entry` widget in a Tcl variable and your scripts will be simpler. Changes to the `entry` widget are reflected in the Tcl variable value, and changes in the Tcl variable are reflected in the `entry` contents.

An `entry` widget has several attributes that control the appearance of the selection and the insert cursor, such as `selectBackground` and `insertWidth`. The `exportSelection` attribute controls whether or not the selected text in the `entry` can be pasted into other applications. The `show` attribute is useful for entries that accept passwords or other sensitive information. Instead of displaying text, asterisks are displayed if `show` is `false`. The `state` attribute determines if the contents of an entry can be modified. Set the `state` to `disabled` to prevent modification, and set it to `normal` to allow modification.

```
.name.entry config -state disabled ;# read-only
.name.entry config -state normal   ;# editable
```

Programming entry widgets

The default bindings for entry widgets are fairly good. However, you can completely control the `entry` with a set of widget operations for inserting, deleting, selecting, and scrolling. The operations involve addressing character positions called *indices*. The indices count from zero. The entry defines some symbolic indices such as `end`. The index corresponding to an X coordinate is specified with @*xcoord*, such as `@26`. Table 16–3 lists the formats for indices.

Table 16–3 Indices for `entry` widgets

`0`	Index of the first character.
`anchor`	The index of the anchor point of the selection.
`end`	Index of the last character.
number	Index a character, counting from zero.
`insert`	The character right after the insertion cursor.
`sel.first`	The first character in the selection.
`sel.last`	The character just after the last character in the selection.
`@`*xcoord*	The character under the specified X coordinate.

Table 16–4 summarizes the widget operations. In the table, `$w` is an `entry` widget.

Table 16–4 Operations on `entry` widgets.

`$w cget` *`option`*	Return the value of the configuration option.
`$w configure ...`	Query or modify the widget configuration.
`$w delete` *`first`* `?`*`last`*`?`	Delete the characters from *`first`* to *`last`*, not including the character at *`last`*. The character at *`first`* is deleted if *`last`* is not given.
`$w get`	Return the string in the entry.
`$w icursor` *`index`*	Move the insert cursor.
`$w index` *`index`*	Return the numerical index corresponding to *index*.
`$w insert` *`index string`*	Insert the *`string`* at the given *`index`*.
`$w scan mark` *`x`*	Start a scroll operation. *`x`* is a screen coordinate.
`$w scan dragto` *`x`*	Scroll from previous mark position.
`$w select adjust` *`index`*	Move the boundary of an existing selection.
`$w select clear`	Clear the selection.
`$w select from` *`index`*	Set the anchor position for the selection.
`$w select present`	Returns 1 if there is a selection in the entry.
`$w select range` *`start end`*	Select the characters from *`start`* to the one just before *`end`*.
`$w select to` *`index`*	Extend a selection.
`$w xview`	Return the offset and span of visible contents. These are both real numbers between 0 and 1.0
`$w xview` *`index`*	Shift the display so the character at `index` is at the left edge of the display.
`$w xview moveto` *`fraction`*	Shift the display so that *`fraction`* of the contents are off the left edge of the display.
`$w xview scroll` *`num what`*	Scroll the contents by the specified number of *`what`*, which can be `units` or `pages`.

Use the bind interface from Chapter 13 to browse the `Entry` class bindings. You will see examples of these operations. For example, the binding for `<Button-1>` includes the following commands.

```
%W icursor @%x
%W select from @%x
if {[lindex [%W config -state] 4] == "normal"} {focus %W}
```

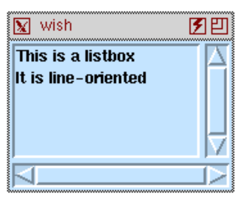
Recall that the % triggers substitutions in binding commands, and that %W is replaced with the widget pathname and %x is replaced with the X coordinate of the mouse event. Chapter 13 describes bindings and these substitutions in detail. These commands set the insert point to the point of the mouse click by using the @%x index, which will be turned into something like @17 when the binding is invoked. The binding also starts a selection. If the entry is not in the disabled state, then keyboard focus is given to the entry so that it gets KeyPress events.

The listbox Widget

The listbox widget displays a set of text lines in a scrollable display. The basic text unit is a line. There are operations to insert, select, and delete lines, but there are no operations to modify the characters in a line. As such, the listbox is suitable for displaying a set of choices, such as in a file selection dialog, but it is not right for a general purpose text editor. The text widget described in the next chapter is designed for general text display and editing.

A listbox is almost always associated with a scrollbar, even though you can also scroll by dragging with the middle mouse button. The following example associates two scrollbars with a listbox, one for both the X and Y directions.

Example 16–2 A listbox with scrollbars.




```
proc ScrolledListbox { parent args } {
	# Create listbox attached to scrollbars, pass thru $args
	frame $parent
	eval {listbox $parent.list \
		-yscrollcommand [list $parent.sy set] \
		-xscrollcommand [list $parent.sx set]} $args
	# Create scrollbars attached to the listbox
	scrollbar $parent.sx -orient horizontal \
		-command [list $parent.list xview]
	scrollbar $parent.sy -orient vertical \
		-command [list $parent.list yview]
	# Arrange them in the parent frame
	pack $parent.sx -side bottom -fill x
	pack $parent.sy -side right -fill y
	# Pack to allow for resizing
	pack $parent.list -side left -fill both -expand true
	return $parent.list
}
ScrolledListbox .f -width 20 -height 5 -setgrid true
```

```
pack .f -fill both -expand true
.f.list insert end "This is a listbox"
.f.list insert end "It is line-oriented"
```

The ScrolledListbox procedure uses the eval and $args technique described in Chapter 6 to pass through extra arguments to the listbox. The example specifies the width, height, and setgrid values for the listbox. The main window becomes resizable as a side effect of gridding the listbox. Chapter 24 describes gridding and geometry in more detail.

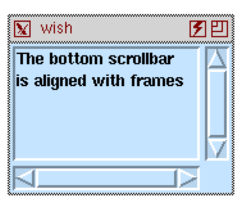
The listbox has two scrolling commands associated with it, one each for the X and Y directions. These commands set the parameters of the scrollbar with its set command. This is most of what you need to know about scrollbars, although Chapter 15 describes them in more detail.

The listbox is controlled by the command associated with a scrollbar. When the user clicks on the scrollbar, it commands the listbox to change its display. When the listbox changes its display, it commands the scrollbars to update their display. Thus the scrollbars display themselves correctly whether the user scrolls with the scrollbars or by dragging the listbox with the middle mouse button.

The list command is used to construct the scroll commands so that $parent gets expanded and the command has the right form. It is also used in the scrollbar commands when defining their command attributes. While you could use double-quotes instead of list here, make a habit of using list when constructing Tcl commands. This habit prevents bugs that arise when variable values include special characters. For more discussion, see Chapter 6 and 3.

The packing commands arrange three widgets on three different sides of the parent frame. This is one of the few cases where a mixture of horizontal and vertical packing within the same frame works. However, the arrangement causes the bottom scrollbar to extend past the listbox a little bit. If you want to line up the bottom scrollbar with the listbox, you must introduce a little frame to space things out, and then another frame to hold this spacer and one of the scrollbars. The second version of ScrolledListbox presented below achieves this.

Example 16–3 A listbox with scrollbars and better alignment.




```
proc ScrolledListbox2 { parent args } {
    # Create listbox attached to scrollbars, pass thru $args
    eval {listbox $parent.list \
        -yscrollcommand [list $parent.sy set]
        -xscrollcommand [list $parent.pad.sx set]} $args
```

```
    scrollbar $parent.sy -orient vertical
        -command [list $parent.list yview]
    # Create extra frame to hold pad and horizontal scrollbar
    frame $parent.pad
    scrollbar $parent.pad.sx -orient horizontal
        -command [list $parent.list xview]
    # Create padding based on the scrollbar's width
    # and its borders.
    set pad [expr [$parent.sy cget -width] + 2* \
        ([$parent.sy cget -bd] + \
         [$parent.sy cget -highlightthickness])]
    frame $parent.pad.it -width $pad -height $pad
    # Arrange everything in the parent frame
    pack $parent.pad -side bottom -fill x
    pack $parent.pad.it -side right
    pack $parent.pad.sx -side bottom -fill x
    pack $parent.sy -side right -fill y
    pack $parent.list -side left -fill both -expand true
    return $parent.list
}
ScrolledListbox2 .f -width 20 -height 5 -setgrid true
pack .f -expand true -fill both
.f.list insert end \
    "The bottom scrollbar" "is aligned with frames"
```

The packing parameters are a bit subtle in `ScrolledListbox2`. The bottom scrollbar of the previous example is replaced by a frame, `$parent.pad`, that contains the horizontal scrollbar and another frame for padding. It is packed with the same parameters that the horizontal scrollbar was packed with before: `-side bottom -fill x`. The padding frame and the horizontal scrollbar are packed inside that. Here we see another case of mixing horizontal and vertical packing, with the pad to the right and the scrollbar to the bottom:

```
pack $parent.pad.sx -side bottom -fill x
```

The combination of `-side bottom` and `-fill x` enables the scrollbar to fill out the whole bottom side of the virtual packing cavity . Another way to pack the horizontal scrollbar is given below . The `-expand true` is required, otherwise the `-side left` squeezes down the scrollbar to a minimum size.

```
pack $parent.pad.sx -side left -fill x -expand true
```

Programming listboxes

The listbox is the first of the specialized text widgets that really requires some programming to make it useful. There are listbox operations to insert and delete items. There are also a set of operations to control the selection and scrolling, but these are already used by the pre-defined bindings, which are discussed in the next section.

The `listbox` operations use indices to reference lines in the listbox. The lines are numbered starting at zero. Keyword indices are also used for some special lines. The listbox keeps track of an `active` element, which is displayed with underlined text. There is also a selection anchor that is used when adjusting selections. The keywords used for indices are summarized in the table below .

Table 16–5 Indices for `listbox` widgets

`0`	Index of the first line.
`active`	The index of the activated line.
`anchor`	The index of the anchor point of the selection.
`end`	Index of the last line.
`number`	Index a line, counting from zero.
`@`*`x,y`*	The line closest to the specified X and Y coordinate.

The table below gives the operations used to program a listbox. In the table , `$w` is a `listbox` widget. Most of the operations have to do with the selection, and these operations are already programmed by the default bindings for the `Listbox` widget class.

Table 16–6 Operations on `listbox` widgets..

`$w activate` *`index`*	Activate the specified line.
`$w bbox` *`index`*	Return the bounding box of the text in the specified line in the form: `xoff yoff width height`.
`$w cget` *`option`*	Return the value of the configuration option.
`$w configure ...`	Query or modify the widget configuration.
`$w curselection`	Return a list of indices of the selected lines.
`$w delete` *`first`* `?`*`last`*`?`	Delete the lines from *`first`* to *`last`*, including the line at *`last`*. The line at *`first`* is deleted if *`last`* is not given.
`$w get` *`first`* `?`*`last`*`?`	Return the lines from *`first`* to *`last`* as a list.
`$w index` *`index`*	Return the numerical index corresponding to *`index`*.
`$w insert` *`index ?string string string ...?`*	Insert the *`string`* items before the line at *`index.`* If *`index`* is `end`, then append the items.
`$w nearest` *`y`*	Return the index of the line closest to the widget-relative Y coordinate.
`$w scan mark` *`x y`*	Start a scroll operation. *`x`* and *`y`* are widget-relative screen coordinates
`$w scan dragto` *`x y`*	Scroll from previous mark position.
`$w see` *`index`*	Adjust the display so the line at *`index`* is visible.

Table 16–6 Operations on `listbox` widgets..

`$w select anchor` *`index`*	Anchor the selection at the specified line.
`$w select clear`	Clear the selection.
`$w select includes` *`index`*	Returns 1 if the line at *`index`* is in the selection.
`$w select set` *`start ?end?`*	Select the lines from *`start`* to *`end`*.
`$w xview`	Return the offset and span of visible contents. These are both real numbers between 0 and 1.0
`$w xview` *`index`*	Shift the display so the character at *`index`* is at the left edge of the display.
`$w xview moveto` *`fraction`*	Shift the display so that *`fraction`* of the contents are off the left edge of the display.
`$w xview scroll` *`num what`*	Scroll the contents horizontally by the specified number of *`what`*, which can be `units` or `pages`.
`$w yview`	Return the offset and span of visible contents. These are both real numbers between 0 and 1.0
`$w yview` *`index`*	Shift the display so the line at *`index`* is at the top edge of the display.
`$w yview moveto` *`fraction`*	Shift the display so that *`fraction`* of the contents are off the top of the display.
`$w yview scroll` *`num what`*	Scroll the contents vertically by the specified number of *`what`*, which can be `units` or `pages`.

The most common programming task for a listbox is to insert text. If your data is in a list, then you can loop through the list and insert each element at the end.

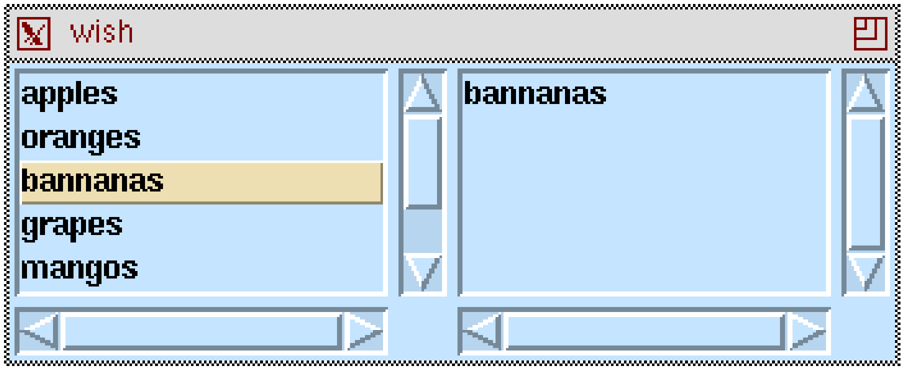
```
foreach item $list {
    $listbox insert end $item
}
```

You can do the same thing by using `eval` to concatenate the list onto a single `insert` command.

```
eval {$listbox insert end} $list
```

It is also common to react to mouse clicks on a listbox. The following example displays two listboxes. When the user clicks on an item in the first listbox, it is copied into the second listbox. When an item in the second listbox is selected, it is removed.

Example 16–4 Choosing items from a `listbox`




```
proc ListSelect { parent choices } {
    # Create two lists side by side
    frame $parent
    ScrolledListbox2 $parent.choices -width 20 -height 5 \
        -setgrid true
    ScrolledListbox2 $parent.picked -width 20 -height 5 \
        -setgrid true
    # The setgrid allows interactive resizing, so the
    # pack parameters need expand and fill.
    pack $parent.choices $parent.picked -side left \
        -expand true -fill both

    # Selecting in choice moves items into picked
    bind $parent.choices.list <ButtonPress-1> \
        {ListSelectStart %W %y}
    bind $parent.choices.list <B1-Motion> \
        {ListSelectExtend %W %y}
    bind $parent.choices.list <ButtonRelease-1> \
        [list ListSelectEnd %W %y $parent.picked.list]

    # Selecting in picked deletes items
    bind $parent.picked.list <ButtonPress-1> \
        {ListSelectStart %W %y}
    bind $parent.picked.list <B1-Motion> \
        {ListSelectExtend %W %y}
    bind $parent.picked.list <ButtonRelease-1> \
        {ListDeleteEnd %W %y}

    # Insert all the choices
    # eval is used to construct a command where each
    # item in choices is a separate argument
    eval {$parent.choices.list insert 0} $choices
}

proc ListSelectStart { w y } {
    $w select anchor [$w nearest $y]
}
```

```
proc ListSelectExtend { w y } {
    $w select set anchor [$w nearest $y]
}
proc ListSelectEnd {w y list} {
    $w select set anchor [$w nearest $y]
    foreach i [$w curselection] {
        $list insert end [$w get $i]
    }
}
proc ListDeleteEnd {w y} {
    $w select set anchor [$w nearest $y]
    foreach i [lsort -decreasing [$w curselection]] {
        $list delete $i
    }
}
ListSelect .f {apples oranges bananas \
            grapes mangos peaches pears}
pack .f -expand true -fill both
```

The `ListSelect` procedure creates two lists using `ScrolledListbox2`. Bindings are created to move items from choices to picked, and to delete items from picked. Consider the `<ButtonRelease-1>` binding for choices:

```
bind $parent.choices.list <ButtonRelease-1> \
    [list ListSelectEnd %W %y $parent.picked.list]
```

The `list` command is used to construct the Tcl command because we need to expand the value of `$parent` at the time the binding is created. The command will be evaluated later at the global scope, and `parent` will not be defined after the `ListSelect` procedure returns. Or, worse yet, an existing global variable named `parent` will be used, which is unlikely to be correct!

Short procedures are used to implement the binding command, even though two of them are just one line. This style has two advantages. First, it confines the `%` substitutions done by `bind` to a single command. Second, if there are any temporary variables, such as the loop counter `i`, they are hidden within the scope of the procedure.

The `ListSelectEnd` procedure extends the current selection to the `listbox` item under the given Y coordinate. It gets the list of all the selected items, and loops over this list to insert them into the other list. The `ListDeleteEnd` procedure is similar. However, it sorts the selection indices in reverse order. It deletes items from the bottom up so the indices remain valid throughout the process.

Listbox Bindings

A `listbox` has an active element and it may have one or more selected elements. The active element is highlighted with an underline, and the selected elements are highlighted with a different color. There are 4 selection modes for a listbox, and the bindings vary somewhat depending what mode the listbox is in. You can always select items with the mouse bindings, but the listbox needs the input focus for the key bindings to work. The 4 possible selectMode settings are described below .

Table 16–7 The values for the selectMode of a listbox.

single	A single element can be selected.
browse	A single element can be selected, and the selection can be dragged with the mouse. This is the default.
multiple	More than one element can be selected by toggling the selection state of items, but you only select or deselect one line at a time.
extended	More than one element can be selected by dragging out a selection with the shift or control keys.

Browse select mode

In browse selection mode, <Button-1> selects the item under the mouse and dragging with the mouse moves the selection, too. Table 16–8 gives the bindings for browse mode.

Table 16–8 Bindings for browse selection mode.

<Button-1>	Select the item under the mouse. This becomes the active element, too.
<B1-Motion>	Same as <Button-1>, the selection moves with the mouse.
<Shift-Button-1>	Activate the item under the mouse. The selection is not changed.
<Key-Up> <Key-Down>	Move the active item up (down) one line, and select it.
<Control-Home>	Activate and select the first element of the listbox.
<Control-End>	Activate and select the last element of the listbox.
<space> <Select> <Control-slash>	Select the active element.

Single select mode

In single selection mode, <Button-1> selects the item under the mouse, but dragging the mouse does not change the selection. When you release the mouse, the item under that point is activated. Table 16–9 gives the bindings for single mode.

Table 16–9 Bindings for a listbox in single selectMode.

<ButtonPress-1>	Select the item under the mouse.

Table 16–9 Bindings for a `listbox` in `single` selectMode.

`<ButtonRelease-1>`	Activate the item under the mouse.
`<Shift-Button-1>`	Activate the item under the mouse. The selection is not changed.
`<Key-Up> <Key-Down>`	Move the active item up (down) one line. The selection is not changed.
`<Control-Home>`	Activate and select the first element of the listbox.
`<Control-End>`	Activate and select the last element of the listbox.
`<space> <Select>` `<Control-slash>`	Select the active element.
`<Control-backslash>`	Clear the selection.

Extended select mode

In `extended` selection mode multiple items are selected by dragging out a selection with the first mouse button. Hold down the `Shift` key to adjust the ends of the selection. Use the `Control` key to make a disjoint selection. The `Control` key works in a toggle fashion, changing the selection state of the item under the mouse. If this starts a new part of the selection, then dragging the mouse extends the new part of the selection. If the toggle action cleared the selected item, then dragging the mouse continues to clear the selection. The extended mode is quite intuitive once you try it out. Table 16–10 gives the complete set of bindings for `extended` mode.

Table 16–10 Bindings for `extended` selection mode.

`<Button-1>`	Select the item under the mouse. This becomes the anchor point for adjusting the selection.
`<B1-Motion>`	Sweep out a selection from the anchor point.
`<ButtonRelease-1>`	Activate the item under the mouse.
`<Shift-Button-1>`	Adjust the selection from the anchor item to the item under the mouse.
`<Shift-B1-Motion>`	Continue to adjust the selection from the anchor .
`<Control-Button-1>`	Toggle the selection state of the item under the mouse, and make this the anchor point.
`<Control-B1-Motion>`	Set the selection state of the items from the anchor point to the item under the mouse to be the same as the selection state of the anchor point.
`<Key-Up> <Key-Down>`	Move the active item up (down) one line, and start out a new selection with this item as the anchor point.

Table 16–10 Bindings for `extended` selection mode.

`<Shift-Up> <Shift-Down>`	Move the active element up (down) and extend the selection to include this element.
`<Control-Home>`	Activate and select the first element of the listbox.
`<Control-Shift-Home>`	Extend the selection to the first element.
`<Control-End>`	Activate and select the last element of the listbox.
`<Control-Shift-End>`	Extend the selection to the last element.
`<space> <Select>`	Select the active element.
`<Escape>`	Cancel the previous selection action.
`<Control-slash>`	Select everything in the listbox.
`<Control-backslash>`	Clear the selection.

Multiple select mode

In `multiple` selection mode you can have more than one item selected, but you only add or remove one item at a time. Dragging the mouse does not sweep out a selection. If you click on a selected item it is deselected. Table 16–11 gives the complete set of bindings for `multiple` selection mode.

Table 16–11 Bindings for `multiple` selection mode.

`<Button-1>`	Select the item under the mouse.
`<ButtonRelease-1>`	Activate the item under the mouse.
`<Key-Up> <Key-Down>`	Move the active item up (down) one line, and start out a new selection with this item as the anchor point.
`<Shift-Up> <Shift-Down>`	Move the active element up (down).
`<Control-Home>`	Activate and select the first element of the listbox.
`<Control-Shift-Home>`	Activate the first element of the listbox.
`<Control-End>`	Activate and select the last element of the listbox.
`<Control-Shift-End>`	Activate the last element of the listbox.
`<space> <Select>`	Select the active element.
`<Control-slash>`	Select everything in the listbox.
`<Control-backslash>`	Clear the selection.

Scroll bindings

There are a number of bindings that scroll the display of the listbox. As well as the standard middle-drag scrolling, there are some additional key bindings for scrolling. The scroll-related bindings are summarized in the table below.

Table 16–12 Scroll bindings common to all selection modes.

`<Button-2>`	Mark the start of a scroll operation.
`<B2-Motion>`	Scroll vertically *and* horizontally.
`<Left> <Right>`	Scroll horizontally by one character.
`<Control-Left> <Control-Right>` `<Control-Prior> <Control-Next>`	Scroll horizontally by one screen width.
`<Prior> <Next>`	Scroll vertically by one screen height.
`<Home> <End>`	Scroll to left and right edges of the screen, respectively.

listbox attributes

Table 16–13 lists the `listbox` widget attributes. The table uses the X resource name for the attribute, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 16–13 Resource names of attributes for `listbox` widgets.

`background`	Background color (also `bg`).
`borderWidth`	Extra space around the edge of the text.
`cursor`	Cursor to display when mouse is over the widget.
`exportSelection`	If `true`, then the selected text is exported via the X selection mechanism.
`font`	Font for the text.
`foreground`	Foreground color. (Also `fg`).
`height`	Number of lines in the listbox.
`highlightColor`	Color for input focus highlight.
`highlightThickness`	Thickness of focus highlight rectangle.
`relief`	3D relief: `flat`, `sunken`, `raised`, `groove`, `ridge`.
`selectBackground`	Background color of selection.
`selectForeground`	Foreground color of selection.
`selectBorderWidth`	Widget of selection border. Non-zero for 3D effect.
`selectMode`	`browse`, `single`, `extended`, `multiple`
`setGrid`	Boolean. Set gridding attribute.

Table 16–13 Resource names of attributes for `listbox` widgets.

`width`	Width, in average character sizes.
`xScrollCommand`	Used to connect `listbox` to a horizontal scrollbar.
`yScrollCommand`	Used to connect `listbox` to a vertical scrollbar.

Geometry gridding

The `setGrid` attribute affects interactive resizing of the window containing the listbox. By default, a window can be resized to any size. If gridding is turned on, however, the size is restricted so that a whole number of listbox lines and a whole number of average-width characters will be displayed. In addition, gridding affects the user feedback during an interactive resize, assuming the window manager displays the current size of the window in numeric terms. Without gridding the size is reported in pixel dimensions. When gridding is turned on, then the size is reported in grided units (e.g., `20x10`).

CHAPTER 17

Focus, Grabs, and Dialogs

Input focus directs keyboard events to different widgets. The grab mechanism lets a widget capture the input focus. Dialog boxes are the classic example of a user interface object that uses grabs.

Dialog boxes are a classic feature in a user interface. The application needs some user response before it can continue. It displays some information and some controls, and the user must interact with this dialog box before the application can continue. To implement this, the application grabs the input focus so the user can only interact with the dialog box. This chapter describes focus and grabs, and finishes with some examples of dialog boxes.

Input Focus

The X window system directs keyboard events to the main window that currently has the input focus. The application, in turn, directs the keyboard events to one of the widgets within that toplevel window. The `focus` command is used to set focus to a particular widget. Tk remembers what widget has focus within a toplevel window, and automatically gives focus to that widget when the window manager gives focus to a toplevel window.

Two focus models are used: focus-follows-mouse, and click-to-type. In the first, moving the mouse into a toplevel window gives the application focus. In the second, the user must click on a window for it to get focus, and thereafter the position of the mouse is not important. Within a toplevel window, Tk uses the click-to-type model by default. In addition, the creation order of widgets determines a traversal order for focus. Use the `tk_focusNext` and `tk_focusPrev` procedures to change the focus to the next (previous) widget in the focus order.

You can get the focus-follows-mouse model within a toplevel window by calling the `tk_focusFollowsMouse` procedure. However, in many cases you will find that an explicit focus model is actually more convenient for users.

The focus command

Table 17–1 sumarises the `focus` command. The implementation supports an application that has windows on multiple displays with a separate focus window on each display. The -displayof option can be used to query the focus on a particular display. The -lastfor option finds out what widget last had the focus within the same toplevel as another window. Tk will restore focus to that window if the widget that has the focus is destroyed. The toplevel widget gets the focus if no widget claims it.

Table 17–1 The `focus` command.

`focus`	Return the widget that currently has the focus on the display of the application's main window.
`focus` *`window`*	Set the focus to *`window`*.
`focus -displayof` *`win`*	Return the focus widget on the same display as *`win`*.
`focus -lastfor` *`win`*	Return the name of the last widget to have the focus on the display of *`win`*.

Focus follows mouse

To implement the focus-follows-mouse model you need to track the `<Enter>` and `<Leave>` events that are generated when the mouse moves in and out of widgets. The `tk_focusFollowsMouse` procedure sets up this binding (the real procedure is only slightly more complicated).

```
bind all <Enter> {focus %W}
```

It might be better to set up this binding only for those widget classes for which it makes sense to get the input focus. The next example does this. The focus detail (`%d`) is checked by the code in order to filter out extraneous focus events generated by X. That trick is borrowed from `tk_focusFollowsMouse`. (The Xlib reference manual discourages you from attempting to understand the details of its focus mechanism. After reading it, I understand why. This code seems plausible.)

Example 17–1 Setting focus-follows-mouse input focus model.

```
proc FocusFollowsMouse {} {
    foreach class {Button Checkbutton Radiobutton Menubutton\
                Menu Canvas Entry Listbox Text} {
```

```
            bind $class <Enter> {
               if {("%d" == "NotifyAncestor") ||
                   ("%d" == "NotifyNonlinear") ||
                   ("%d" == "NotifyInferior")} {
                  focus %W
               }
            }
         }
      }
   }
```

Click to type

To implement the click-to-type focus model you need to set up a binding on the button events. The <Any-Button> event will work nicely.

```
bind all <Any-Button> {focus %W}
```

Again, it might be better to restrict this binding to those classes for which it makes sense. The previous example can be modified easily to account for this.

Hybrid models

You can develop hybrid models that are natural for users. If you have a dialog or form-like window with several entry widgets, then it can be tedious for the user to position the mouse over the various entries in order to direct focus. Instead, click-to-type as well as keyboard shortcuts like `<Tab>` or `<Return>` may be easier for the user, even if they use focus-follows-mouse with their window manager.

Grabbing the Focus

An input *grab* is used to override the normal focus mechanism. For example, a dialog box can grab the focus so that the user cannot interact with other windows in the application. The typical scenario is that the application is performing some task but it needs user input. The grab restricts the user's actions so it cannot drive the application into an inconsistent state. A *global grab* prevents the user from interacting with other applications, too, even the window manager. Tk menus use a global grab, for example, which is how they unpost themselves no matter where you click the mouse. When an application prompts for a password a global grab is also a good idea. This prevents the user from accidentally typing their password into a random window. Table 17–1 summarizes the `grab` command.

Table 17–2 The `grab` command.

`grab ?-global? window`	Set a grab to a particular window.

Table 17–2 The `grab` command.

`grab current ?window?`	Query the grabs on the display of *window*, or on all displays if *window* is omitted.
`grab release window`	Release a grab on *window*.
`grab set ?-global? win`	Set a grab to a particular window.
`grab status window`	Returns `none`, `local`, or `global`.

In most cases you only need to use the `grab` and `grab release` commands. Note that the `grab set` command is equivalent to the `grab` command. The next section includes examples that use the grab command.

Dialogs

The tkwait Command

This section presents a number of different examples of dialogs. In nearly all cases the `tkwait` command is used to wait for the dialog to complete. This command waits for something to happen, and the key thing is that `tkwait` allows events to be processed while waiting. This effectively suspends part of your application while other parts can respond to user input. Table 17–1 summarizes the `tkwait` command.

Table 17–3 The `tkwait` command.

`tkwait variable varname`	Wait for the global variable *varname* to be set.
`tkwait visibility win`	Wait for the window *win* to become visible.
`tkwait window win`	Wait for the window *win* to be destroyed.

The variable specified in the `tkwait variable` command is a global variable. Remember this if you use procedures to modify the variable. They must declare it global or the tkwait command will not notice the assignments.

The tkwait visibility waits for the visibility state of the window to change. Most commonly this is used to wait for a newly created window to become visible. For example, if you have any sort of animation in a complex dialog, you'll want to wait until the dialog is displayed before starting the animation.

Prompter dialog

The `GetValue` dialog gets a value from the user, returning the value entered, or the empty string if the user cancels the operation.

Example 17–2 A simple dialog.


```
proc GetValue { prompt } {
    global prompt
    set f [toplevel .prompt -borderwidth 10]
    message $f.msg -text $prompt
    entry $f.entry -textvariable prompt(result)
    set b [frame $f.buttons -bd 10]
    pack $f.msg $f.entry $f.buttons -side top -fill x

    bind $f.entry <Return> {set prompt(ok) 1}
    bind $f.entry <Control-c> {set prompt(ok) 0}
    button $b.ok -text OK -command {set prompt(ok) 1}
    button $b.cancel -text Cancel -command {set prompt(ok) 0}

    focus $f.entry
    grab $w
    tkwait variable prompt(ok)
    grab release $w
    destroy $w
    if {$prompt(ok)} {
        return $prompt(result)
    } else {
        return {}
    }
}
GetValue "Please enter a name"
```

The `tkwait variable` command is used to wait for the dialog to complete. Anything that changes the `prompt(ok)` variable will cause the `tkwait` command to return, and then the dialog will be completed. The variable is set if the user presses the `OK` or `Cancel` buttons, of if they press `<Return>` or `<Control-c>` in the entry widget.

The focus is set to the entry widget and a grab is placed so the user can only interact with the dialog box. The sequence of `focus`, `grab`, `tkwait`, and `grab release` is fairly standard.

Destroying widgets

The `destroy` command deletes one or more widgets. If the widget has children, all the children are destroyed, too. The example deletes the dialog with a single destroy operation on the toplevel window .

You can wait for a window to be deleted with the tkwait window command.

```
tkwait window pathname
```

This provides an alternate way to synchronize things when using dialogs.

Focusing on buttons

The previous example defined two key bindings for the entry widget that invoked the same commands used by the buttons. An alternative that is more like the interfaces in the W indows environment is to has key bindings that shift the focus to different widgets. The Tk widgets, even buttons and scrollbars, have bindings that support keyboard interaction. A `<space>` for example, will invoke the command associated with a button, assuming the button has the input focus. The Tk widgets highlight themselves when they get focus, too, so the user has some notion of what is going on.

The following bindings cause the `<Tab>` key to cycle focus among the widgets in the prompter dialog.

```
bind $f.entry <Tab> [list focus $b.ok]
bind $b.ok <Tab> [list focus $b.cancel]
bind $b.cancel <Tab> [list focus $f.entry]
```

Another way to shift focus is to use a standard key sequence where the last letter indicates what widget to focus on. The label and button widgets have an underline attribute that indicates what letter to underline. If you use that letter as the ID for a widget, users will know (with some training) how to focus on different widgets.

Animation with the update command

Suppose you want to entertain your user while your application is busy . By default, the user interface will just hang until your processing completes. Even if you are changing a label or entry widget in the middle of processing, the updates to that widget will be batched up until an idle moment. The user will notice that the window is not refreshed when it gets obscured and uncovered, and they will not see your feedback. The `update` command forces Tk to go through its event loop. The safest way to use update is with its `idletasks` option.

Example 17–3 A feedback procedure.

```
proc Feedback { message } {
	global feedback
	# An entry widget is used because it won’t change size
	# based on the message length, and it can be scrolled
```

```
    set e $feedback(entry)
    $e config -state normal
    $e delete 0 end
    $e insert 0 $message
    # Leave the entry in a read-only state
    $e config -state disabled
    # Force a display update
    update idletasks
}
```

The Tk widgets update their display at idle moments, which basically means after everything else is taken care of. This lets them collapse updates into one interaction with the X server, and it improves the batching effects that are part of the X protocol. A call to `update idletasks` causes any pending display updates to be processed.

If you use the update command with no options, then all events are processed. In particular, user input events are processed. If you are not careful, it can have unexpected effects because another thread of execution is launched into your Tcl interpreter. The current thread is suspended and any callbacks that result from input events get to execute. It is usually better to use the `tkwait` command instead of a naked `update`.

File Selection Dialog

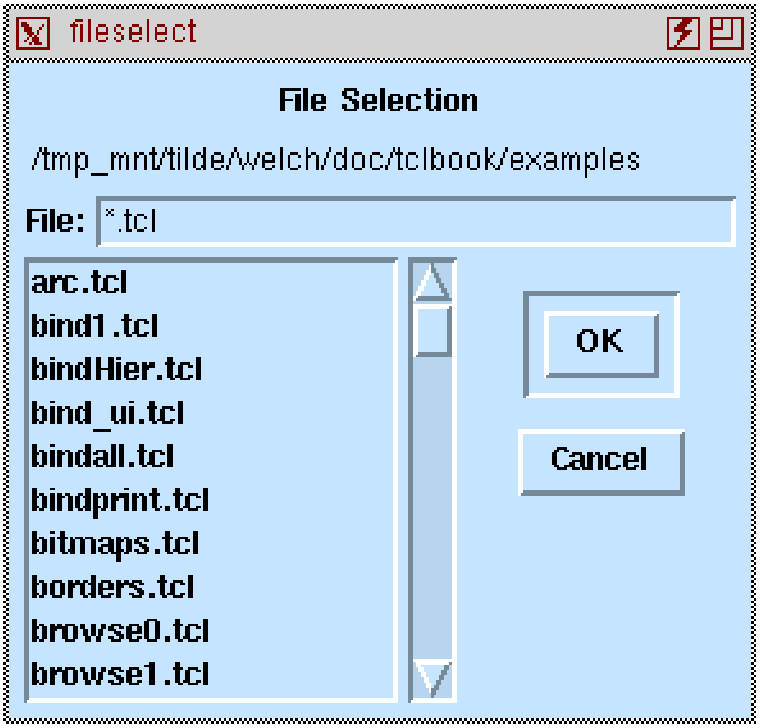
Selecting files is common to many applications. This section presents a file selection dialog that supports file name completion. The dialog displays the current directory, and has an `entry` widget in which to enter a name. It uses a `listbox` to display the contents of the current directory. There is an `OK` and a `Cancel` button. These buttons set a variable and the dialog finishes, returning the selected pathname or an empty string.

Some key bindings are set up to allow for keyboard selection. Once the correct pathname is entered, a `<Return>` is equivalent to hitting the `OK` button. `<Control-c>` is equivalent to hitting the `Cancel` button. `<space>` does file name completion, which is discussed in more detail below. A `<Tab>` changes focus to the listbox so that its bindings can be used for selection. The arrow keys move the selection up and down. A `<space>` copies the current name into the entry, and a `<Return>` is like hitting the `OK` button. A picture of the dialog appears below.

Creating the dialog

Example 17–4 A file selection dialog.

```
proc fileselect {{why "File Selection"} {default {}} } {
    global fileselect

    catch {destroy .fileselect}
```




```
set t [toplevel .fileselect -bd 4]

message $t.msg -aspect 1000 -text $why
pack $t.msg -side top -fill x

# Create a read-only entry for the current directory
set fileselect(dirEnt) [entry $t.dir -width 15 \
   -relief flat -state disabled]
pack $t.dir -side top -fill x

# Create an entry for the pathname
# The value is kept in fileselect(path)
frame $t.top
label $t.top.l -text "File:" -padx 0
set e [entry $t.top.path -relief sunken \
   -textvariable fileselect(path)]
pack $t.top -side top -fill x
pack $t.top.l -side left
pack $t.top.path -side right -fill x -expand true
set fileselect(pathEnt) $e

# Set up bindings to invoke OK and Cancel
bind $e <Return> fileselectOK
bind $e <Control-c> fileselectCancel
bind $e <space> fileselectComplete
focus $e
```

```
# Create a listbox to hold the directory contents
listbox $t.list -yscrollcommand [list $t.scroll set]
scrollbar $t.scroll -command [list $t.list yview]

# A single click copies the name into the entry
# A double-click selects the name
bind $t.list <Button-1> {fileselectClick %y}
bind $t.list <Double-Button-1> {
   fileselectClick %y ; fileselectOK
}

# Warp focus to listbox so the user can use arrow keys
bind $e <Tab> "focus $t.list ; $t.list select set 0"
bind $t.list <Return> fileselectTake
bind $t.list <space> {fileselectTake ; break}
bind $t.list <Tab> "focus $e"

# Create the OK and Cancel buttons
# The OK button has a rim to indicate it is the default
frame $t.buttons -bd 10
frame $t.buttons.ok -bd 2 -relief sunken
button $t.buttons.ok.b -text OK \
   -command fileselectOK
button $t.buttons.cancel -text Cancel \
   -command fileselectCancel

# Pack the list, scrollbar, and button box
# in a horizontal stack below the upper widgets
pack $t.list -side left -fill both -expand true
pack $t.scroll -side left -fill y
pack $t.buttons -side left -fill both
pack $t.buttons.ok $t.buttons.cancel \
   -side top -padx 10 -pady 5
pack $t.buttons.ok.b -padx 4 -pady 4

# Initialize variables and list the directory
if {[string length $default] == 0} {
   set fileselect(path) {}
   set dir [pwd]
} else {
   set fileselect(path) [file tail $default]
   set dir [file dirname $default]
}
set fileselect(dir) {}
set fileselect(done) 0

# Wait for the listbox to be visible so
# we can provide feedback during the listing
tkwait visibility .fileselect.list
fileselectList $dir

tkwait variable fileselect(done)
destroy .fileselect
return $fileselect(path)
```

```
}
```

The `tkwait` command is used in two ways in this dialog. First, `tkwait visibility` is used so we can delay listing the directory until the listbox is visible. A special message is displayed there during the listing, which can take some time for larger directories. This ensures that the dialog appears quickly so the user knows what is going on. After this, `tkwait variable` is used to wait for the user interaction to complete.

Listing the directory

Much of the complexity of the file selection process has to do with looking through the directory and doing file name completion. The code uses `file dirname` to extract the directory part of a pathname, and `file tail` to extract the last component of a directory . The `glob` command is used to do the directory listing. It also has the nice property of expanding pathnames that begin with a ~.

Example 17–5 Listing a directory for fileselect.

```
proc fileselectList { dir {files {}} } {
    global fileselect

    # Update the directory, being careful
    # to view the tail end
    set e $fileselect(dirEnt)
    $e config -state normal
    $e delete 0 end
    $e insert 0 $dir
    $e config -state disabled
    $e xview moveto 1

    # Give the user some feedback
    set fileselect(dir) $dir
    .fileselect.list delete 0 end
    .fileselect.list insert 0 Listing...
    update idletasks

    .fileselect.list delete 0
    if {[string length $files] == 0} {
        # List the directory and add an
        # entry for the parent directory
        set files [glob -nocomplain $fileselect(dir)/*]
        .fileselect.list insert end ../
    }

    # Sort the directories to the front
    set dirs {}
    set other {}
    foreach f [lsort $files] {
        if [file isdirectory $f] {
            lappend dirs [file tail $f]/
```

```
        } else {
            lappend others [file tail $f]
        }
    }
    foreach f [concat $dirs $others] {
     .fileselect.list insert end $f
    }
}
```

The fileselectList procedure does three things: update the directory name, provide some feedback, and finally list the directory. The entry widget that holds the pathname is kept read-only, so it has to be reconfigured when it is updated. Then the entry's xview moveto operation is used to ensure that the tail end of the pathname is visible. An argument of 1 is specified, which tries to scroll all of the pathname off screen to the left, but the widget implementation limits the scrolling so it works just the way we want it to.

Before the directory is listed the listbox is cleared an a single line is put into it. The update idletasks command forces Tk to do all its pending screen updates so the user sees this message while she waits for the directory listing.

The directory listing itself is fairly straight-forward. The glob command is used if no file list is passed in. The slow part is the file isdirectory test on each pathname to decide if the trailing / should be appended. This requires a file system stat system call, which can be expensive. The directories are sorted into the beginning of the list.

Accepting a name

There are a few different cases when the user clicks the OK button or otherwise accepts the current name. The easy case is when the name matches an existing file in the current directory. The complete name is put into the global fileselect(path) variable and the fileselect(done) variable is set to signal that we are done. If an existing directory is specified, then the fileselectList routine is called to list its contents. The parent directory has to be handled specially because we want to trim off the last component of the current directory. Without the special case the ../ components remain in the directory pathname, which is still valid, but ugly.

The user can type a file name pattern, such as *.tcl. We test for this by trying the glob command to see if it matches on anything. If it matches a single file, fileselectOK is called recursively. Otherwise, fileselectList is called to display the results.

If the glob fails, then the user may have typed in a new absolute pathname. Until now we have assumed that what they typed was relative to the current directory. The glob command is used again, and we leave out the current directory. If glob fails the user has not typed anything good, so we attempt file name completion. Otherwise we ignore the return from glob and call fileselectOK recursively instead. This works because we fix up the current directory with file dirname, which doesn't care if the input name exists.

Example 17–6 Accepting a file name.

```
proc fileselectOK {} {
    global fileselect

    # Handle the parent directory specially
    if {[regexp {˙˙/?} $fileselect(path)]} {
        set fileselect(path) {}
        fileselectList [file dirname $fileselect(dir)]
        return
    }

    set path $fileselect(dir)/$fileselect(path)

    if [file isdirectory $path] {
        set fileselect(path) {}
        fileselectList $path
        return
    }
    if [file exists $path] {
        set fileselect(path) $path
        set fileselect(done) 1
        return
    }
    # Neither a file or a directory.
    # See if glob will find something
    if [catch {glob $path} files] {
        # No, perhaps the user typed a new
        # absolute pathname
        if [catch {glob $fileselect(path)} path] {
            # Nothing good - attempt completion
            fileselectComplete
            return
        } else {
            # OK - try again
            set fileselect(dir) \
                [file dirname $fileselect(path)]
            set fileselect(path) \
                [file tail $fileselect(path)]
            fileselectOK
            return
        }
    } else {
        # Ok - current directory is ok,
        # either select the file or list them.
        if {[llength [split $files]] == 1} {
            set fileselect(path) $files
            fileselectOK
        } else {
            set fileselect(dir) \
                [file dirname [lindex $files 0]]
            fileselectList $fileselect(dir) $files
        }
    }
```

```
}
```

Easy stuff

If the user hits `Cancel`, or presses `<Control-c>`, the result variable is cleared and the done variable is set to end the dialog. The `fileselectCancel` procedure does this.

The user can select something from the listbox in two ways. If they click on an item, then the listbox `nearest` operation is used to find out which one. If they have shifted focus to the listbox with `<Tab>` and then press `<space>`, then the listbox `curselection` operation is used to find out what is selected. These two operations return a listbox index, so the listbox `get` operation is used to get the actual value.

Example 17–7 Simple support routines.

```
proc fileselectCancel {} {
    global fileselect
    set fileselect(done) 1
    set fileselect(path) {}
}

proc fileselectClick { y } {
    # Take the item the user clicked on
    global fileselect
    set l .fileselect.list
    set fileselect(path) [$l get [$l nearest $y]]
    focus $fileselect(pathEnt)
}
proc fileselectTake {} {
    # Take the currently selected list item and
    # change focus back to the entry
    global fileselect
    set l .fileselect.list
    set fileselect(path) [$l get [$l curselection]]
    focus $fileselect(pathEnt)
}
```

File name completion

File name completion tries to match what the user has typed against existing files. It more complex than using `glob` to match files because the common prefix of the matching names is filled in for the user . In addition, the matching names are listed in the `listbox`. The search for the matching prefix is crude, but effective. The prefix begins as the string typed by the user . Then, the first matching name from the `glob` is used as the source for the rest of the prefix. The prefix is lengthened by one until it fails to match all the names in the list.

Example 17–8 File name completion.

```
proc fileselectComplete {} {
	global fileselect

	# Do file name completion
	# Nuke the space that triggered this call
	set fileselect(path) [string trim $fileselect(path) \t\ ]

	# Figure out what directory we are looking at
	# dir is the directory
	# tail is the partial name
	if {[string match /* $fileselect(path)]} {
		set dir [file dirname $fileselect(path)]
		set tail [file tail $fileselect(path)]
	} elseif [string match ~* $fileselect(path)] {
		if [catch {file dirname $fileselect(path)} dir] {
			return	;# Bad user
		}
		set tail [file tail $fileselect(path)]
	} else {
		set path $fileselect(dir)/$fileselect(path)
		set dir [file dirname $path]
		set tail [file tail $path]
	}
	# See what files are there
	set files [glob -nocomplain $dir/$tail*]
	if {[llength [split $files]] == 1} {
		# Matched a single file
		set fileselect(dir) $dir
		set fileselect(path) [file tail $files]
	} else {
		if {[llength [split $files]] > 1} {
			# Find the longest common prefix
			set l [expr [string length $tail]-1]
			set miss 0
			# Remember that files has absolute paths
			set file1 [file tail [lindex $files 0]]
			while {!$miss} {
				incr l
				if {$l == [string length $file1]} {
					# file1 is a prefix of all others
					break
				}
				set new [string range $file1 0 $l]
				foreach f $files {
					if ![string match $new* [file tail $f]] {
						set miss 1
						incr l -1
						break
					}
				}
			}
			set fileselect(path) [string range $file1 0 $l]
		}
```

```
        fileselectList $dir $files
    }
}
```

CHAPTER 18

The text Widget

Tk `text` widget is a general purpose editable text widget with features for line spacing, justification, tags, marks, and embedded windows.

The Tk text widget is a versatile widget that is simple to use for basic text display and manipulation, while at the same time it has many advanced features to support sophisticated applications. The line spacing and justification can be controlled on a line-by-line basis. Fonts, sizes, and colors are controlled with *tags* that apply to ranges of text. Edit operations use positional *marks* that keep track of locations in text, even as text is inserted and deleted.

Text widget taxonomy

Tk provides several widgets that handle text. The `label` widget provides a single line of read-only text. The `entry` widget provides a single line for user type-in. The `message` widget arranges its read-only text in multiple lines with a given width or aspect ratio. The `listbox` widget holds a set of scrollable text lines. And, finally, the `text` widget is a general-purpose multi-line text widget. While it is possible to use the `text` widget for all of these purposes, using the specialized widgets can be more convenient.

The main drawback of having several different text-related widgets is that there is some inconsistency among the widgets. The `entry`, `listbox`, and `text` widgets have different notions of text addressing. The `entry` addresses characters, the `listbox` addresses lines, and the `text` widget addresses lines and characters. In addition, both the `entry` and `text` widgets provide operations to insert and edit text, but they differ slightly. Chapter 18 describes the `entry` and `listbox` widgets. Chapter 15 describes the `label` and `message` widgets.

Text Indices

The characters in a text widget are addressed by their line number and the character position within the line. Lines are numbered starting at one, while characters are numbered starting at zero. The numbering for lines was chosen to be compatible with other programs that number lines starting at one, like compilers that generate line-oriented error messages. Here are some examples of text indices.

1.0	The first character.
1.1	The second character on the first line.
1.end	The character just before the newline on line one.

There are also symbolic indices. The `insert` index is the position at which new characters are normally inserted when the user types in characters. You can define new indices called *marks*, too, as described below. Table 18–1 summarizes the various forms for a text index.

Table 18–1 Forms for the indices in `text` widgets.

`line.char`	Lines count from 1. Characters count from 0.
`@`*`x,y`*	The character under the specified position.
`end`	Just after the very last character.
`insert`	The position right after the insert cursor.
`mark`	Just after the named *`mark`*.
`tag``.first`	The first character in the range tagged with *`tag`*.
`tag``.last`	Just after the last character tagged with *`tag`*.
`window`	The position of the embedded *`window`*.

The text widget supports a simple sort of arithmetic on indices. You can specify "the end of the line with this index" and "three characters before this index", and so on. This is done by grouping a modifying expression with the index. For example, the `insert` index can be modified like this:

```
"insert lineend"
"insert -3 chars"
```

Table 18–2 summarizes the set of index modifiers.

The interpretation of indices and their modifiers is designed to operate well with the `delete` and `addtag` operations of the `text` widget. These operations

Table 18–2 Index modifiers for `text` widgets.

`+ count chars`	*count* characters past the index.
`- count chars`	*count* characters before the index.
`+ count lines`	*count* lines past the index, retaining character position.
`- count lines`	*count* lines past the index, retaining character position.
`linestart`	The beginning of the line.
`lineend`	The end of the line (just before the newline character).
`wordstart`	The first character of a word.
`wordend`	Just after the last character of a word.

apply to a range of text defined by two indices. The second index refers to the character just after the end of the range. For example, the following command deletes the word containing the insert cursor .

```
$t delete "insert wordstart" "insert wordend"
```

You can supply several modifiers to an index, and they are applied in left to right order . If you want to delete a whole include, including the trailing newline, you need to do the following. Otherwise the newline remains and you are left with a blank line.

```
$t delete "insert linestart" "insert lineend +1 char"
```

Text Marks

A mark is a symbolic name for a position between two characters. Marks have the property that when text is inserted or deleted they retain their logical position, not their numerical index position. Even if you delete the text surrounding a mark it remains intact. Marks are created with the `mark set` operation, and have to be explicitly deleted with the `mark unset` operation. Once defined, a mark can be used in operations that require indices.

```
$t mark set foobar "insert wordstart"
$t delete foobar "foobar lineend"
$t mark unset foobar
```

When a mark is defined, it is set to be just before the character specified by the index. In the example above, this is just before the first character of the word where the insert cursor is. When a mark is used in an operation that requires an index it refers to the character just after the mark. So, in many ways the mark seems associated with the character right after it, except that the mark remains even if that character is deleted.

You can use almost any string for the name of a mark. However , do not use pure numbers, and do not include spaces, plus (+) or minus (-). These characters are used in the mark arithmetic and will cause problems if you put them into mark names. The `mark names` operation returns a list of all defined marks.

The `insert` mark defines where the insert cursor is displayed. The `insert` mark is treated specially: you cannot remove it with the `mark unset` operation. Attempting to do so does not raise an error, though, so the following is a quick way to unset all marks.

```
eval {$t mark unset} [$t mark names]
```

Each mark has a *gravity* that determines what happens when characters are inserted at the mark. The default gravity is `right`, which means that the mark sticks to the character that was to its right. Inserting text at a mark with `right` gravity causes the mark to be pushed along so it is always after the inserted text.* With `left` gravity the mark stays with the character to its left, so inserted text goes after the mark and the mark does not move. The `mark gravity` operation is used to query and modify the gravity of a mark.

```
$t mark gravity foobar
=> right
$t mark gravity foobar left
```

Text Tags

A tag is a symbolic name for a range of characters. You can use almost any string for the name of a tag. However, do not use pure numbers, and do not include spaces, plus (+) or minus (-). These characters are used in the mark arithmetic and will cause problems if you use them tag names.

A tag has attributes that affect the display of text that is tagged with it. These attributes include fonts, colors, line spacing and justification. A tag can have event bindings so you can create hypertext. A tag can be used for non-display reasons, too. The text widget operations described later include operations to find out what tags are defined and where they are applied.

A tag is added to a range with the `tag add` operation. The following command applies the tag `everywhere` to all the text in the widget.

```
$t tag add everywhere 1.0 end
```

You can add one or more tags when text is inserted, too.

```
$t insert insert "new text" someTag someOtherTag
```

If you do not specify tags when text is inserted, then the text will pick up any tags that are present on the characters on both sides of the insertion point. (Before Tk 4.0, tags from the left hand character were picked up.) If you specify tags in the `insert` operation, only those tags are applied to the text.

A tag is removed from a range of text with the `tag remove` operation. Even if there is no text labeled with a tag, its attribute settings are remembered. All information about a tag can be removed with the `tag delete` operation.

* In versions of Tk before 4.0, marks only had right gravity, which made some uses of marks awkward.

```
$t tag remove everywhere 3.0 6.end
$t tag delete everywhere
```

Tag attributes

The attributes for a tag are defined with the `tag configure` operation. Table 18–3 gives the complete set of attributes for tags. For example, a tag for blue text is defined with the following command:

```
$t tag configure blue -foreground blue
```

Table 18–3 Attributes for `text` tags.

`-background` *color*	The background color for text.
`-bgstipple` *bitmap*	A stipple pattern for the background color.
`-borderwidth` *pixels*	The width for 3D border effects.
`-fgstipple` *bitmap*	A stipple pattern for the foreground color.
`-font font`	The font for the text.
`-foreground color`	The foreground color for text.
`-justify how`	`left right center`
`-lmargin1` *pixels*	Normal left indent for a line.
`-lmargin2` *pixels*	Indent for the part of a line that gets wrapped.
`-offset` *pixels*	Baseline offset. Positive for superscripts.
`-relief` *what*	`flat raised sunken ridge groove`
`-rmargin` *pixels*	Right hand margin.
`-spacing1` *pixels*	Additional space above a line.
`-spacing2` *pixels*	Additional space above wrapped part of line.
`-spacing3` *pixels*	Additional space below a line.
`-underline` *boolean*	If true, the text is underlined.

The relief and border width attributes go together. If you specify a relief without a borderwidth, then there is no visible effect. The default relief is `flat`, too, so if you specify a borderwidth without a relief you won't see any effect either.

The stipple attributes require a bitmap argument. For example, to "grey out" text you could use a foreground stipple of `gray50`. Bitmaps and colors are explained in more detail in Chapter 23.

```
$t tag configure disabled -fgstipple gray50
```

You can set up the appearance (and bindings) for tags once in your application, even before you have labeled any text with the tags. The attributes are retained until you explicitly remove the tag. If you are going to use the same appearance over and over again then it will be more efficient to do the setup once so that Tk can retain the graphics context.

The next example defines a few tags for character styles you might see in an editor. The example is a bit over simplified. In practice you would want to parameterize the font family and the size for the fonts.

Example 18–1 Tag configurations for basic character styles.

```
proc TextStyles { t } {
    $t tag configure bold -font *-times-bold-r-*-12-*
    $t tag configure italic -font *-times-medium-i-*-12-*
    $t tag configure fixed -font fixed
    $t tag configure underline -underline true
    $t tag configure super -offset 6 \
        -font *-helvetica-medium-r-*-8-*
    $t tag configure sub -offset -6 \
        -font *-helvetica-medium-r-*-8-*
}
```

On the other hand, if you change the configuration of a tag, any text with that tag will be redisplayed with the new attributes. Similarly, if you change a binding on a tag, all tagged characters are affected immediately.

Mixing attributes from different tags

A character can be labeled with more than one tag. In this case an ordering among the tags determines the priority of the attributes from the different tags. The tags might not conflict, either. For example, one tag could determine the font, another could determine that foreground color, and so on. Only if different tags try to supply the same attribute is the priority ordering taken into account. The latest tag added to a range of text has the highest priority. The ordering of tags can be controlled explicitly with the `tag raise` and `tag lower` commands.

You can achieve interesting effects by composing attributes from different tags. In a mail reader, for example, the listing of messages in a mail folder can use one color to indicate messages that are marked for delete, and it can use another color for messages that are marked to be moved into another folder. These tags might be defined like this:

```
$t tag configure deleted -background grey75
$t tag configure moved -background yellow
```

These tags don't mix. However, a selection could be indicated with an underline, for example.

```
$t tag configure select -underline true
```

With these tags defined, you can add and remove the `select` tag to indicate what messages have been selected, and the underline is independent of the background color.

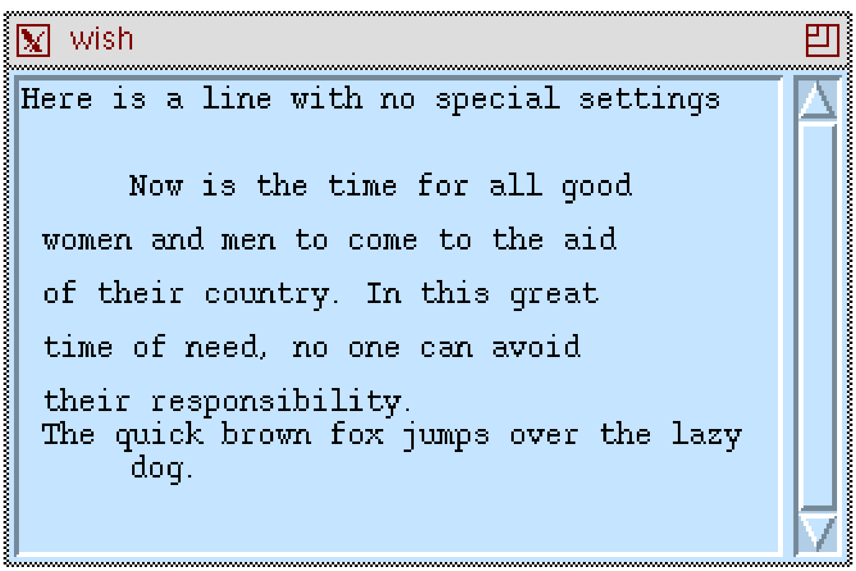
Line Spacing and Justification

The spacing and justification for text has several attributes. The situation is complicated a little by wrapped text lines. The text widget distinguishes between the first *display line* and the remaining display lines for a given text line. For example, if a line in the text widget has 80 characters but the window is only wide enough for 30, then the line may be wrapped onto three display lines. See Table 18–7 on page 226 for a description of the text widget's `wrap` attribute that controls this behavior.

Spacing is controlled with three attributes, and there are global spacing attributes as well as per-tag spacing attributes. The `-spacing1` attribute adds space above the first display line, while `-spacing2` adds space above the subsequent display lines that exist because of wrapping, if any. The `-spacing3` attribute adds space before the last display line, which could be the same as the first display line if the line is not wrapped at all.

The margin settings also distinguish between the first and remaining display lines. The `-lmargin1` attribute specifies the indent for the first display line, while the `-lmargin2` attribute specifies the indent for the rest of the display lines, if any. There is only a single attribute, `-rmargin`, for the right indent. These margin attributes are only tag attributes. The closest thing for the text widget as a whole is the `-padx` attribute, but this adds an equal amount of spacing on both sides.

Example 18–2 Line spacing and justification in the `text` widget.




```
proc TextExample { f } {
    frame $f
    pack $f -side top -fill both -expand true
```

```
    set t [text $f.t -setgrid true -wrap word \
        -width 42 -height 14 \
        -yscrollcommand "$f.sy set"]
    scrollbar $f.sy -orient vert -command "$f.t yview"
    pack $f.sy -side right -fill y
    pack $f.t -side left -fill both -expand true

    $t tag configure para -spacing1 0.25i -spacing2 0.1i \
        -lmargin1 0.5i -lmargin2 0.1i -rmargin 0.5i
    $t tag configure hang -lmargin1 0.1i -lmargin2 0.5i

    $t insert end "Here is a line with no special settings\n"
    $t insert end "Now is the time for all good women and men
to come to the aid of their country. In this great time of
need, no one can avoid their responsibility.\n"
    $t insert end "The quick brown fox jumps over the lazy
dog."

    $t tag add para 2.0 2.end
    $t tag add hang 3.0 3.end
}
```

The example defines two tags, `para` and `hang`, that have different spacing and margins. The `-spacing1` setting for para causes the white space after the first line. The `-spacing2` setting causes the white space between the wrapped portions of the second paragraph. The `hang` tag has no spacing attributes so the last paragraph starts right below the previous paragraph. You can also see the difference between the `-lmargin1` and `-lmargin2` settings.

The newline characters are inserted explicitly. Each newline character defines a new line for the purposes of indexing, but not necessarily for display, and this example shows. In the third line there is no newline. This means that if more text is inserted at the `end` mark, it will be on the same logical line.

The values for the spacing and margin parameters are in screen units. Because different fonts are different sizes, you may need to compute the spacings as a function of the character sizes. The `bbox` operation returns the bounding box for a given character.

```
$t insert 1.0 "ABCDE"
$t bbox 1.0
=> 3 3 7 13
set height [lindex [$t bbox 1.0] 3]
=> 13
```

Text justification is limited to three styles: `left`, `right` or `center`. There is no setting that causes the text to line up on both margins, which would have to be achieved by introducing variable spacing between words.

The Selection

The selection is implemented with a predefined tag named `sel`. When the user makes a selection with the mouse, the selected characters are tagged with `sel`. If the application tags characters with `sel` then the selection is changed to be those characters. This is just what is done with the default bindings in order to set the selection.

The `exportSelection` attribute of a text widget controls whether or not selected text is exported by the X selection mechanism. By default the selection is exported. In this case, when another widget or application asserts ownership of the selection then the `sel` tag is removed from any characters that are tagged with it.

You cannot delete the `sel` tag with the `tag delete` operation. However, it is not an error to do so. You can delete all the tags on the text widget with the following command:

```
eval {$t tag delete} [$t tag names]
```

Tag Bindings

A tag can have bindings associated with it so that when the user clicks on different areas of the text display then different things happen. The syntax for the `tag bind` command is similar to that of the main Tk `bind` command. You can both query and set the bindings for a tag. Chapter 13 describes the `bind` command and the syntax for events in detail.

The only events supported by the `tag bind` command are `Enter`, `Leave`, `ButtonPress`, `Motion`, and `KeyPress`. `ButtonPress` and `KeyPress` can be shorted to `Button` and `Key` as in the regular `bind` command. The `Enter` and `Leave` events are triggered when the mouse moves in and out of characters with a tag, which is different than when the mouse moves in and out of the window.

If a character has multiple tags, then the bindings associated with all the tags will be invoked, in the order from lowest priority tag to highest priority tag. After all the tag bindings have run, the binding associated with the main widget is run, if any. The `continue` and `break` commands work inside tag bindings in a similar fashion as they work with regular command bindings. See Chapter 13 for the details.

The next example defines a text button that has a highlighted relief and an action associated with it.

Example 18–3 An active `text` button.

```
proc TextButton { t start end command } {
    global textbutton
    if ![info exists textbutton(uid)] {
        set textbutton(uid) 0
    } else {
        incr textbutton(uid)
```

```
    }
    set tag button$textbutton(uid)
    $t tag configure $tag -relief raised -borderwidth 2
    if {[tk colormodel $t] == "color"} {
        $t tag configure $tag -background thistle
    }
    $t tag bind $tag <Enter> {%W config -cursor tcross}
    $t tag bind $tag <Leave> {
        %W config -cursor [lindex [%W config -cursor] 3]
    }
    $t tag bind $tag <Button-1> $command
    $t tag add $tag $start $end
}
```

The example generates a new tag name so that each text button is unique. The relief and background are set for the tag to set it apart visually. The `tk colormodel` command is used to find out if the display supports color before adding a colored background to the tag. The command is bound to `<Button-1>`, which is the same as `<ButtonPress-1>`. The cursor is changed when the mouse is over the tagged are by binding to the `<Enter>` and `<Leave>` events. Upon leaving the tagged area, the cursor is reset to the default setting for the widget, which is the third element of the configuration information. Another approach would be to save and restore the cursor setting for the window.

To behave even more like a button the action should trigger upon `<ButtonRelease-1>`, and the appearance should change upon `<ButtonPress-1>`. If this is important to you, you can always embed a real Tk button. Embedding widgets is described in the next section.

Embedded Widgets

The text widget can display one or more other widgets as well as text. You can include picture, for example, by constructing it in a canvas and then inserting the canvas into the text widget. An embedded widget takes up one character in terms of indices. You can address the widget by its index position or by the Tk pathname of the widget.

For example, suppose `$t` names a text widget. The following commands create a button and insert it into the text widget. The button behaves normally, and in this case it will invoke the `Help` command when the user clicks on it.

```
button $t.help -bitmap questhead -command Help
$t window create end -window $t.help
```

By default an embedded widget is centered vertically on its text line. You can adjust this with the `-align` option to the `window create` command. The possible alignments are `top`, `center`, `baseline`, or `bottom`.

`top`	Top of widget lines up with top of text line.
`center`	Center of widget lines up with center of text line.

`baseline`	Bottom of widget lines up with text baseline.
`bottom`	Bottom of widget lines up with bottom of text line.

You can postpone the creation of the embedded widget by specifying a Tcl command that creates the window, instead of specifying the -window option. This delayed creation is useful if you have lots of widgets embedded in your text. In this case the Tcl command is evaluated just before the text widget needs to display the window. In other words, when the user scrolls the text so the widget would appear, the Tcl command is run to create the widget.

Example 18–4 Delayed creation of embedded widgets.

```
$t window create end -create [list MakeGoBack $t]
proc MakeGoBack { t } {
    button $t.goback -text "Go to Line 1" \
        -command [list $t see 1.0]
}
```

It might seem excessive to introduce the `MakeGoBack` procedure in this example, but it actually makes things easier. The fact that the button has its own command make that you have to quote things if you do not introduce the procedure. Furthermore, if you are really creating buttons on the fly they are likely to require more complex setup than in this simple example. Without the procedure you have to do the following. It may not seem messy now, but if you need to execute more than one Tcl command to create the widget or if the embedded button has a complex command, the quoting can quickly get out of hand.

```
$t window create end -create "button $t.goback \
    -text {Go to Line 1} -command \{$t.goback see 1.0\}"
```

Table 18–4 gives the complete set of options for creating embedded widgets. You can change these later with the `window configure` operation. For example:

```
$t window configure $t.goback -align bottom.
```

Table 18–4 Options to the `window create` operation.

`-align` *`where`*	`top center bottom baseline`
`-create` *`command`*	Tcl command to create the widget.
`-padx` *`pixels`*	Padding on either side of the widget.
`-pady` *`pixels`*	Padding above and below the widget.
`-stretch` *`boolean`*	If true, the widget is stretched vertically to match the spacing of the text line.
`-window` *`pathname`*	Tk pathname of the widget to embed.

You can specify the window to reconfigure with either the index where the window is located, or by its pathname. Note that end is not a good candidate for the index because the text widget treats it specially . A character , or widget, inserted at end is really inserted right before the very last character , which is always a newline.

Text Bindings

There is an extensive set of default bindings for text widgets. In general, the commands that move the insert cursor also clear and the selection. Often you can hold the Shift key down to extend the selection instead, or hold the Control key down to move the insert cursor without affecting the insert cursor . Table 18–5 lists the default bindings for the text widget.

Table 18–5 Bindings for the text widget.

<Any-Key>	Insert normal printing characters.
<Button-1>	Set the insert point, clear the selection, set focus.
<Control-Button-1>	Set the insert point without affecting the selection.
<B1-Motion>	Sweep out a selection from the insert point.
<Double-Button-1>	Select the word under the mouse.
<Triple-Button-1>	Select the line under the mouse.
<Shift-Button-1>	Adjust the end of selection closest to the mouse.
<Shift-B1-Motion>	Continue to adjust the selection.
<Button-2>	Paste the selection, or set the scrolling anchor .
<B2-Motion>	Scroll the window .
<Key-Left> <Control-b>	Move the cursor left one character . Clear selection.
<Shift-Left>	Move the cursor and extend the selection.
<Control-Left>	Move the cursor by words. Clear the selection.
<Control-Shift-Left>	Move the cursor by words. Extend the selection.
<Key-Right> <Control-f>	All Right bindings are analogous to Left bindings.
<Meta-b> <Meta-f>	Same as <Control-Left> and <Control-Right>
<Key-Up> <Control-p>	Move the cursor up one line. Clear the selection.
<Shift-Up>	Move the cursor up one line. Extend the selection.
<Control-Up>	Move the cursor up by paragraphs, which are a group of lines separated by a blank line.
<Control-Shift-Up>	Move the cursor up by paragraph. Extend selection.

Table 18–5 Bindings for the `text` widget.

`<Key-Down> <Control-n>`	All `Down` bindings are analogous to `Up` bindings.
`<Next> <Prior>`	Move the cursor by a screenful. Clear the selection.
`<Shift-Next>` `<Shift-Prior>`	Move the cursor by a screenful. Extend the selection.
`<Home> <Control-a>`	Move the cursor to line start. Clear the selection.
`<Shift-Home>`	Move the cursor to line start. Extend the selection.
`<End> <Control-e>`	Move the cursor to line end. Clear the selection.
`<Shift-End>`	Move the cursor to line end. Extend the selection.
`<Control-Home>` `<Meta-less>`	Move the cursor to the beginning of text. Clear the selection.
`<Control-End>` `<Meta-greater>`	Move the cursor to the end of text. Clear the selection.
`<Select>` `<Control-space>`	Set the selection anchor to the position of the cursor.
`<Shift-Select>` `<Control-Shift-space>`	Adjust the selection to the position of the cursor.
`<Control-slash>`	Select everything in the text widget.
`<Control-backslash>`	Clear the selection.
`<Delete>`	Delete the selection, if any. Otherwise delete the character to the right of the cursor.
`<BackSpace> <Control-h>`	Delete the selection, if any. Otherwise delete the character to the left of the cursor.
`<Control-d>`	Delete character to the right of the cursor.
`<Meta-d>`	Delete word to the right of the cursor.
`<Control-k>`	Delete from cursor to end of the line. If you are at the end of line, delete the newline character.
`<Control-o>`	Insert a newline but do not advance the cursor.
`<Control-w>`	Delete the word to the left of the cursor.
`<Control-x>`	Deletes the selection, if any.
`<Control-t>`	Transpose the characters on either side of the cursor.

Text Operations

Table 18–6 below describes the text widget operations, including some that are not discussed in this chapter. In the table, `$t` is a text widget.

Table 18–6 Operations for the `text` widget.

`$t bbox` *`index`*	Return the bounding box of the character at *`index`*. 4 numbers are returned: *`x y width height`*.
`$t cget` *`option`*	Return the value of the configuration option.
`$t compare` *`i1 op i2`*	Perform index comparison. *`ix`* and *`i2`* are indexes . *`op`* is one of: `< <= == >= > !=`
`$t configure ...`	Query or set configuration options.
`$t debug` *`boolean`*	Enable consistency checking for B-tree code.
`$t delete` *`i1 ?i2?`*	Delete from *`i1`* up to, but not including *`i2.`* Just delete the character at *`i1`* if *`i2`* is not specified.
`$t dlineinfo` *`index`*	Return the bounding box, in pixels, of the display for the line containing index. 5 numbers are returned, *`x y width height baseline`*.
`$t get` *`i1 ?i2?`*	Return the text from *`i1`* to *`i2`*, or just the character at *`i1`* if *`i2`* is not specified.
`$t index` *`index`*	Return the numerical value of *`index`*
`$t insert` *`index chars ?tags?`*	Insert *`chars`* at the specified *`index`*. If *`tags`* are specified they are added to the new characters.
`$t mark gravity` *`name ?direction?`*	Query or assign a gravity direction to the mark *`name`*. *`direction`*, if specified, is `left` or `right`.
`$t mark names`	Return a list of defined marks.
`$t mark set` *`name index`*	Define a mark *`name`* at the given *`index`*.
`$t mark unset` *`name1 ?name2 ...?`*	Delete the named mark(s).
`$t scan mark` *`x y`*	Anchor a scrolling operation.
`$t scan dragto` *`x y`*	Scroll based on a new position.
`$t search` *`?switches? pattern index ?varName?`*	Search for text starting at index. The index of the start of the match is returned. The number of characters in the match is stored in varName. Switches are: `-forw, -back, -exact, -regexp, -nowrap, --`
`$t see` *`index`*	Position the view to see *`index`*.
`$t tag add` *`name i1 ?i2?`*	Add the tag to *`i1`* through, but not including *`i2`*, or just the character at i1 if *`i2`* is not given.
`$t tag bind` *`name ?sequence? ?script?`*	Query or define bindings for the tag *`name`*.
`$t tag cget` *`name option`*	Return the value of *`option`* for tag *`name`*.

Table 18–6 Operations for the `text` widget.

`$t tag delete` *`tag1`* `?`*`tag2`* `...?`	Delete information for the named tags.
`$t tag lower` *`tag`* `?`*`before`*`?`	Lower the priority of *`tag`* to the lowest priority or to just below tag *`below`*.
`$t tag names ?`*`index`*`?`	Return the names of the tags at the specified *`index`*, or in the whole widget, sorted from lowest to highest priority.
`$t tag nextrange` *`tag i1`* `?`*`i2`*`?`	Return a list of two indices that are the next range of text with tag that starts at or after *`i1`* and before index *`i2`*, or the end.
`$t tag raise` *`tag`* `?`*`above`*`?`	Raise the priority of *`tag`* to the highest priority, or to just above the priority of *`above`*.
`$t tag ranges` *`tag`*	Return a list describing all the ranges of tag.
`$t tag remove` *`tag i1`* `?`*`i2`*`?`	Remove tag from the range *`i1`* up to, but not including *`i2`*, or just at *`i1`* if *`i2`* is not specified.
`$t window config` *`ir`* `...`	Query or modify the configuration of the embedded window
`$t window create` *`ir`* `?`*`option value ...?`*	Create an embedded window. The configuration options depend on the type of the window.
`$t xview`	Return two fractions between zero and one that describe the amount of text off screen to the left and the amount of text displayed.
`$t xview moveto` *`fraction`*	Position the text so *`fraction`* of the text is off screen to the left.
`$t xview scroll` *`num what`*	Scroll *`num`* of *`what`*, which is *`units`* or *`pages`*.
`$t yview`	Return two fractions between zero and one that describe the amount of text off screen towards the beginning and the amount of text displayed.
`$t yview moveto` *`fraction`*	Position the text so *`fraction`* of the text is off screen towards the beginning.
`$t yview scroll` *`num what`*	Scroll *`num`* of *`what`*, which is *`units`* or *`pages`*.
`$t yview ?-pickplace?` *`ix`*	Obsoleted by the `see` operation, which is similar.
`$t yview` *`number`*	Position line *`number`* at the top of the screen. Obsoleted by the `yview moveto` operation.

Text Attributes

The table below lists the attributes for the text widget. The table uses the X resource Class name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 18–7 Resource names of attributes for `text` widgets.

`background`	Background color (also `bg`).
`borderWidth`	Extra space around the edge of the text.
`cursor`	Cursor to display when mouse is over the widget.
`font`	Default font for the text.
`foreground`	Foreground color. (Also `fg`).
`highlightColor`	Color for input focus highlight border.
`highlightThickness`	Width of highlight border.
`insertBackground`	Color for the insert cursor.
`insertBorderWidth`	Size of 3D border for insert cursor.
`insertOffTime`	Milliseconds insert cursor blinks off.
`insertOnTime`	Milliseconds insert cursor blinks on.
`insertWidth`	Width of the insert cursor.
`padX`	Extra space to the left and right of the text.
`padY`	Extra space above and below the text.
`relief`	3D relief: `flat, sunken, raised, groove, ridge`.
`selectBackground`	Background color of selected text.
`selectForeground`	Foreground color of selected text.
`selectBorderWidth`	Size of 3D border for selection highlight.
`setGrid`	Enable/disable geometry gridding.
`spacing1`	Extra space above each unwrapped line.
`spacing2`	Space between parts of a line that have wrapped.
`spacing3`	Extra space below an unwrapped line.
`state`	Editable (`normal`) or read-only (`disabled`).
`width`	Width, in characters, of the text display.
`wrap`	Line wrap mode: `none char word`
`xScrollCommand`	Tcl command prefix for horizontal scrolling.
`yScrollCommand`	Tcl command prefix for vertical scrolling.

CHAPTER 19

The canvas Widget

This canvas widget provides a general-purpose display that can programmed to display a variety of objects including arcs, images, lines, ovals, polygons, rectangles, text, and embedded windows.Hello,

Canvas widgets are very flexible widgets that can be programmed to display almost anything and to respond in just about any fashion to user input. A canvas displays objects such as lines and images, and each object can be programed to respond to user input, or they can be animated under program control. There are several pre-defined canvas object types. Chapter X describes the a C programming interface for creating new canvas objects. This chapter presents a couple of examples before covering the features of the canvas in more detail.

Hello, World!

A simple exercise for a canvas is to create an object that you can drag around with the mouse. The next example does this.

Example 19–1 The `canvas` Hello, World! example.

```
proc Hello {} {
    # Create and pack the canvas
    canvas .c -width 400 -height 100
    pack .c
    # Create a text object on the canvas
    .c create text 50 50 -text "Hello, World!" -tag movable
    # Bind actions to objects with the movable tag
```

```
    .c bind movable <Button-1> {Mark %x %y %W}
    .c bind movable <B1-Motion> {Drag %x %y %W}
}

proc Mark { x y w } {
    global state
    # Find the object
    set state($w,obj) [$w find closest $x $y]
    set state($w,x) $x
    set state($w,y) $y
}
proc Drag { x y w } {
    global state
    set dx [expr $x - $state($w,x)]
    set dy [expr $y - $state($w,y)]
    $w move $state($w,obj) $dx $dy
    set state($w,x) $x
    set state($w,y) $y
}
```

The example creates a `text` object and gives it a *tag* named `movable`. Tags are discussed in a moment. The first argument after `create` specifies the type, and the remaining arguments depend on the type of object being created. In this case a `text` object needs two coordinates for its location. There is also text, of course, and finally a tag. The complete set of attributes for text objects are given later in this chapter.

```
.c create text 50 50 -text "Hello, World!" -tag movable
```

The `create` operation returns an ID for the object being created, which would have been 1 in this case. However, the code manipulates the canvas objects by specifying a tag instead of an object ID. A tag is a more general handle on canvas objects. Many objects can have the same tag, and an object can have more than one tag. A tag can be (almost) any string; avoid spaces and numbers. Nearly all the canvas operations operate on either tags or object IDs.

The example defines behavior for objects with the `movable` tag. The pathname of the canvas (`%W`) is passed to `Mark` and `Drag` so these procedures could be used on different canvases. The `%x` and `%y` keywords get substituted with the X and Y coordinate of the event.

```
.c bind movable <Button-1> {Mark %x %y %W}
.c bind movable <B1-Motion> {Drag %x %y %W}
```

The `Move` and `Drag` procedures let you drag the object around the canvas. Because they are applied to any object with the `movable` tag, the `Mark` procedure must first find out what object was clicked on. It uses the `find` operation:

```
set state($w,obj) [$w find closest $x $y]
```

The actual moving is done in `Drag` with the `move` operation:

```
$w move $state($w,obj) $dx $dy
```

Try creating a few other object types and dragging them around, too.

```
.c create rect 10 10 30 30 -fill red -tag movable
```

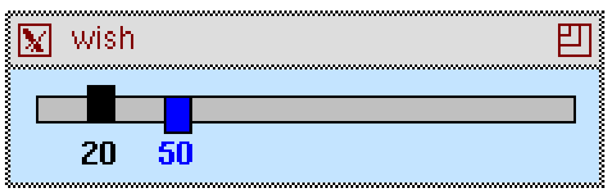
```
.c create line 1 1 40 40 90 60 -width 2 -tag movable
.c create poly 1 1 40 40 90 60 -fill blue -tag movable
```

The example may seem a little cluttered by the general use of state and its indices that are parameterized by the canvas pathname. However, if you get into the habit of doing this early, then you will find it easy to write code that is reusable among programs.

The Double-Slider Example

This section presents an example that constructs a scale-like object that has two sliders on it. The sliders represent the minimum and maximum values for some parameter. Clearly, the minimum cannot be greater than the maximum, and vice versa. The example creates three rectangles on the canvas. One rectangle forms the long axis of the slider. The other two rectangles are markers that represent the values. Two text objects float below the markers to give the current values of the minimum and maximum.

Example 19–2 A double slider `canvas` example.




```
proc Scale2 {w min max {width {}} } {
    global scale2
    if {$width == {}} {
        # Set the long dimension, in pixels
        set width [expr $max - $min]
    }
    # Save parameters
    set scale2($w,scale) [expr ($max-$min)/$width.0]
    set scale2($w,min) $min
    set scale2($w,max) $max
    set scale2($w,Min) $min
    set scale2($w,Max) $max
    set scale2($w,L) 10
    set scale2($w,R) [expr $width+10]

    # Build from 0 to 100, then scale and move it by 10 later
    # Distance between left edges of boxes is 100
    # The left box sticks up, and the right one hangs down
    canvas $w
    $w create rect 0 0 110 10 -fill grey -tag slider
    $w create rect 0 -4 10 10 -fill black -tag {left lbox}
    $w create rect 100 0 110 14 -fill red -tag {right rbox}
    $w create text 5 16 -anchor n -text $min -tag {left lnum}
```

```
    $w create text 105 16 -anchor n -text $max \
        -tag {right rnum} -fill red

    # Stretch/shrink the slider to the right length,
    set scale [expr ($width+10) / 110.0]
    $w scale slider 0 0 $scale 1.0
    # move the right box and text to match new length
    set nx [lindex [$w coords slider] 2]
    $w move right [expr $nx-110] 0

    # Move everything into view
    $w move all 10 10

    # Make the canvas fit comfortably around the image
    set bbox [$w bbox all]
    set height [expr [lindex $bbox 3]+4]
    $w config -height $height -width [expr $width+30]

    # Bind drag actions
    $w bind left  <Button-1> {Start %W %x lbox}
    $w bind right <Button-1> {Start %W %x rbox}
    $w bind left  <B1-Motion> {Move %W %x lbox}
    $w bind right <B1-Motion> {Move %W %x rbox}
}
```

The slider is constructed with absolute coordinates, and then it is scaled to be the desired width. The alternative is to compute the coordinates based on the desired width. The `scale` and `move` operations are used in this example to illustrate them. I also found it a little clearer to use numbers when creating the initial layout as opposed to using `expr` or introducing more variables. It only makes sense to scale the slider bar. If the marker boxes are scaled, then their shape gets distorted, too. The scale operation takes a reference point, which in our case is (0, 0), and independent scale factors for the X and Y dimensions. The scale factor is computed from the `width` parameter, taking into account the extra length added (10) so that the distance between the left edge of the boxes is `$width`.

```
set scale [expr ($width+10) / 110.0]
$w scale slider 0 0 $scale 1.0
```

After stretching the slider bar its new coordinates are used to determine how to move the right box and right hanging text. The `coords` operation returns a list of 4 numbers, *x1 y1 x2 y2*. The distance to move is just the difference between the new right coordinate and the value used when constructing the slider initially. The box and text share the same tag, `right`, so they are both moved with a single `move` command.

```
set nx [lindex [$w coords slider] 2]
$w move right [expr $nx-110] 0
```

After the slider is constructed it is shifted away from (0, 0), which is the upper -left corner of the canvas. The `bbox` operation returns four coordinates, *x1 y1 x2 y2*, that define the bounding box of the items with the given tag. In the example, *y1* is zero, so *y2* gives us the height of the image. The information returned by bbox can be off by a few pixels, and the example needs a few more pixels of height to avoid clipping the text. The width is computed based on the extra length added for the marker box, the 10 pixels the whole image was shifted, and 10 more for the same amount of space on the right side.

```
set bbox [$w bbox all]
set height [expr [lindex $bbox 3]+4]
$w config -height $height -width [expr $width+30]
```

Finally, the bindings are defined for the box and hanging text. Again, the general tags `left` and `right` are used for the bindings. This means you can drag either the box or the text to move the slider. The pathname of the canvas is passed into these procedures so you could have more than one double slider in your interface.

```
$w bind left  <Button-1> {Start %W %x lbox}
$w bind right <Button-1> {Start %W %x rbox}
$w bind left  <B1-Motion> {Move %W %x lbox}
$w bind right <B1-Motion> {Move %W %x rbox}
```

The `Start` and `Move` implementations are shown below.

Example 19–3 Moving the markers for the double-slider.

```
proc Start { w x what } {
   global scale2
   # Remember the anchor point for the drag
   set scale2($w,$what) $x
}
proc Move { w x what } {
   global scale2

   # Compute delta and update anchor point
   set x1 $scale2($w,$what)
   set scale2($w,$what) $x
   set dx [expr $x - $x1]

   # Find out where the boxes are currently
   set rx [lindex [$w coords rbox] 0]
   set lx [lindex [$w coords lbox] 0]

   if {$what == "lbox"} {
      # Constrain the movement to be between the
      # left edge and the right marker.
      if {$lx + $dx > $rx} {
         set dx [expr $rx - $lx]
         set scale2($w,$what) $rx
      } elseif {$lx + $dx < $scale2($w,L)} {
         set dx [expr $scale2($w,L) - $lx]
         set scale2($w,$what) $scale2($w,L)
      }
      $w move left $dx 0

      # Update the minimum value and the hanging text
```

```
        set lx [lindex [$w coords lbox] 0]
        set scale2($w,min) [expr int($scale2($w,Min) + \
            ($lx-$scale2($w,L)) * $scale2($w,scale))]
        $w itemconfigure lnum -text $scale2($w,min)
    } else {
        # Constrain the movement to be between the
        # right edge and the left marker
        if {$rx + $dx < $lx} {
            set dx [expr $lx - $rx]
            set scale2($w,$what) $lx
        } elseif {$rx + $dx > $scale2($w,R)} {
            set dx [expr $scale2($w,R) - $rx]
            set scale2($w,$what) $scale2($w,R)
        }
        $w move right $dx 0

        # Update the maximum value and the hanging text
        set rx [lindex [$w coords right] 0]
        set scale2($w,max) [expr int($scale2($w,Min) + \
            ($rx-$scale2($w,L)) * $scale2($w,scale))]
        $w itemconfigure rnum -text $scale2($w,max)
    }
}
proc Value {w} {
    global scale2
    # Return the current values of the double slider
    return [list $scale2($w,min) $scale2($w,max)]
}
```

The `Start` procedure initializes an anchor position, `scale2($w,$what)`, and `Move` uses this to detect how far the mouse has moved. The change in position, `dx`, is constrained so that the markers cannot move outside their bounds. The anchor is updated if a constraint was used, and this means the marker will not move until the mouse is moved back over the marker. (Try commenting out the assignments to `scale2($w,$what)` inside the `if` statement.) After the marker and hanging text are moved, the value of the associated parameter is computed based on the parameters of the scale. Finally, the `Value` procedure is used to query the current values of the double slider.

The canvas tag facility is very useful. The example uses the `all` tag to move all the items, and to find out the bounding box of the image. The left box and the left hanging text both have the `left` tag. They can be moved together, and they share the same bindings. Similarly, the `right` tag is shared by the right box and the right hanging text. Each item has its own unique tag so it can be manipulated individually, too. Those tags are `slider`, `lbox`, `lnum`, `rbox`, `rnum`. Note that the `itemconfigure` operation is used to change the text. If there were several objects with the same tag, then `itemconfigure` could be used to change all of them.

Canvas Coordinates

The position and possibly the size of a canvas object is determined by a set of coordinates. Different objects are characterized by different numbers of coordinates. For example, text objects have two coordinates, *x1 y1*, that specify their anchor point. A line can have many pairs of coordinates that specify the endpoints of its segments. The coordinates are set when the object is created, and they can be updated later with the `coords` operation. By default coordinates are in pixels. If you suffix a coordinate with one of the following letters then you change these units:

c	centimeters
i	inch
m	millimeters
p	printer points (1/72 inches)

The coordinate space of the canvas positions 0, 0 at the top left corner. Larger X coordinates are to the right, and larger Y coordinates are downward. The `width` and `height` attributes of the canvas determine its viewable area. The `scrollRegion` attribute of the canvas determines the boundaries of the canvas. The view onto the scroll region is changed with the `xview` and `yview` commands. These are designed to work with scrollbars.

Example 19–4 A large scrollable `canvas`.

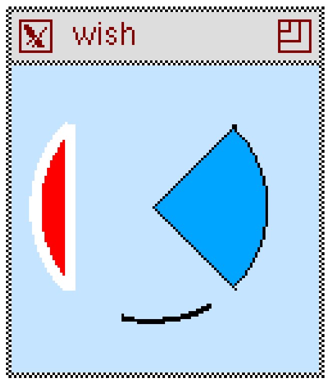
```
proc ScrolledCanvas { c width height region } {
	frame $c
	canvas $c.canvas -width $width -height $height \
		-scrollregion $region \
		-xscrollcommand [list $c.xscroll set] \
		-yscrollcommand [list $c.yscroll set]
	scrollbar $c.xscroll -orient horizontal \
		-command [list $c.canvas xview]
	scrollbar $c.yscroll -orient vertical \
		-command [list $c.canvas yview]
	pack $c.xscroll -side bottom -fill x
	pack $c.yscroll -side right -fill y
	pack $c.canvas -side left -fill both -expand true
	pack $c -side top -fill both -expand true
	return $c.canvas
}
ScrolledCanvas .c 300 200 {0 0 1000 400}
=> .c.canvas
```

The next several sections describe the built-in object types for canvases.

Arcs

An arc is a section of an oval. The dimensions of the oval are determined by four coordinates that are its bounding box. The arc is then determined by two angles, the `start` angle and the `extent`. The region of the oval can be filled or unfilled, and there are three different ways to define the fill region. The `pieslice` style connects the arc with the center point of the oval. The `chord` style connects the two endpoints of the arc. The `arc` style just fills the arc itself and there is no outline.

Example 19–5 Canvas `arc` items.




```
$c create arc 10 10 100 100 -start 45 -extent -90 \
    -style pieslice -fill orange -outline black
$c create arc 10 10 100 100 -start 135 -extent 90 \
    -style chord -fill blue -outline white -width 4
$c create arc 10 10 100 100 -start 255 -extent 45 \
    -style arc -fill black -width 2
```

Table 19–1 gives the complete set of `arc` attributes.

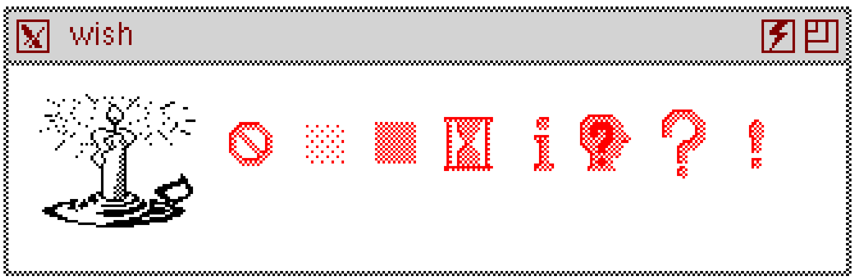
Table 19–1 Attributes for `arc` canvas items.

`-extent` *`degrees`*	The length of the arc in the counter-clockwise direction.
`-fill` *`color`*	The color of the interior of the arc region.
`-outline` *`color`*	The color of the arc itself.
`-start` *`degrees`*	The starting angle of the arc.
`-stipple` *`bitmap`*	A stipple pattern for the fill.
`-style` *`style`*	`pieslice`, `chord`, `arc`
`-tags` *`tagList`*	List of tags for the arc item.
`-width` *`num`*	Width, in canvas coordinates, of the arc and outline.

Bitmap Items

A bitmap is a simple graphic with a foreground and background color . One-bit per pixel is used to choose between the foreground and the background. A canvas `bitmap` item is positioned with two coordinates and an anchor position. Its size is determined by the bitmap data. The `bitmap` itself is specified with a symbolic name or by the name of a file that contains its definition. If the name begins with an @ it indicates a file name. The bitmaps built into wish are shown in the example below . There is a C interface for registering more bitmaps under a name.

Example 19–6 Canvas `bitmap` items.




```
set o [$c create bitmap 10 10 -bitmap @candle.xbm -anchor nw]
set x [lindex [$c bbox $o] 2];# Right edge of bitmap
foreach builtin {error gray25 gray50 hourglass \
               info questhead question warning} {
    incr x 20
    set o [$c create bitmap $x 30 -bitmap $builtin -anchor c\
        -background white -foreground blue]
    set x [lindex [$c bbox $o] 2]
}
```

Table 19–1 gives the complete set of `bitmap` attributes.

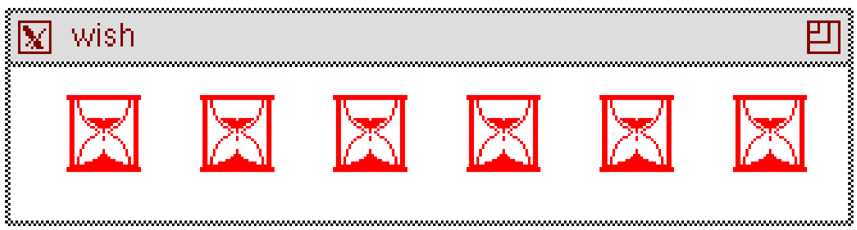
Table 19–2 Attributes for `bitmap` canvas items.

`-anchor` *`position`*	`c n ne e se s sw w nw`
`-background` *`color`*	The background color (for zero bits).
`-bitmap` *`name`*	A built in bitmap.
`-bitmap @`*`filename`*	A bitmap defined by a file.
`-foreground` *`color`*	The foreground color (for one bits).
`-tags` *`tagList`*	List of tags for the bitmap item.

Images

The canvas `image` objects use the general image mechanism of Tk. An image has to be defined first using the `image` command. This command is described Chapter 23 in the section *Bitmaps and Images* . Once you have defined an image, all you need to specify for the canvas is its position, anchor point, and any tags. The size and color information is set when the image is defined. If an image is redefined then anything that is displaying that image gets updated automatically .

Example 19–7 Canvas `image` items.




```
image create bitmap hourglass2 \
   -file hourglass.bitmap -maskfile hourglass.mask \
   -background white -foreground blue]
for {set x 20} {$x < 300} {incr x 20} {
   $c create image $x 10 -image hourglass2 -anchor nw
   incr x [image width hourglass2]
}
```

Table 19–1 lists the attributes for canvas `image` items.

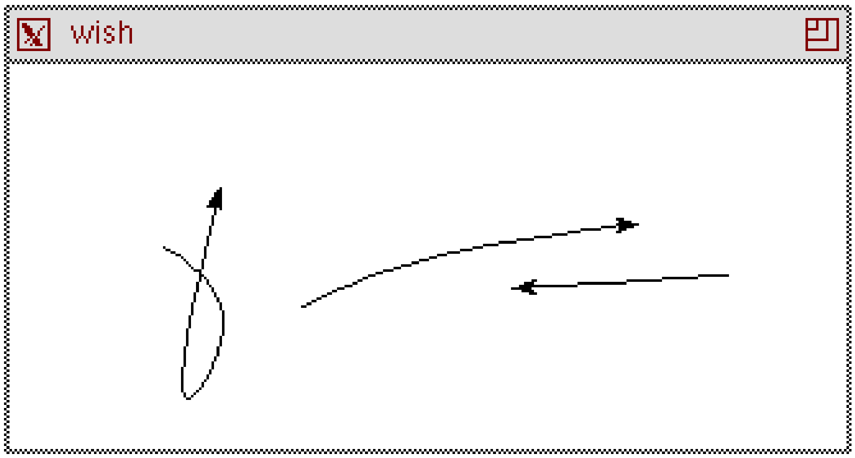
Table 19–3 Attributes for `image` canvas items.

`-anchor` *`position`*	`c n ne e se s sw w nw`
`-image` *`name`*	The name of an image.
`-tags` *`tagList`*	List of tags for the image item.

Line Items

A line has two or more sets of coordinates, where each set of coordinates defines an endpoint of a line segment. The segments can be joined in several different styles, and the whole line can be drawn with a spline fit as opposed to straight-line segments. The next example draws a line in two steps. In the first past, single-segment lines are drawn. When the stroke completes, these are replaced with a single line segment that is drawn with a spline curve.

Example 19–8 A `canvas` stroke drawing example.




```
proc StrokeInit {} {
    canvas .c ; pack .c
    bind .c <Button-1> {StrokeBegin %W %x %y}
    bind .c <B1-Motion> {Stroke %W %x %y}
    bind .c <ButtonRelease-1> {StrokeEnd %W %x %y}
}
proc StrokeBegin { w x y } {
    global stroke
    catch {unset stroke}
    set stroke(N) 0
    set stroke(0) [list $x $y]
}
proc Stroke { w x y } {
    global stroke
    set last $stroke($stroke(N))
    incr stroke(N)
    set stroke($stroke(N)) [list $x $y]
    eval {$w create line} $last {$x $y -tag segments}
}
proc StrokeEnd { w x y } {
    global stroke
    set points {}
    for {set i 0} {$i <= $stroke(N)} {incr i} {
        append points $stroke($i) " "
    }
    $w delete segments
    eval {$w create line} $points \
        {-tag line -joinstyle round -smooth true -arrow last}
}
```

The example uses the `stroke` array to hold the points of the line as it builds up the stroke. At the end of the stroke it assembles the points into a list. The `eval` command is used to splice this list of points into the `create line` command.

Recall that `eval` uses `concat` if it gets multiple argument. The other parts of the `create line` command are protected by braces so they only get evaluated once. Chapter 6 describes this trick in more detail.

The arrow attribute adds an arrow head to the end of the stroke. If you try out this example you'll notice that the arrow isn't always aimed like you expect. This is because there are often many points generated quite close together as you lift up on the mouse button. In fact, the X and Y coordinates of seen by `StrokeEnd` are always the same as those seen by the last `Stroke` call. If you add this duplicate point to the end of the list of points, no arrowhead is drawn at all. In practice you might want to make `Stroke` filter out points that are two close together.

Table 19–1 gives the complete set of line attributes. The `capstyle` affects the way the ends of the line are drawn. The `joinstyle` affects the way line segments are joined together.

Table 19–4 Attributes for `line` canvas items.

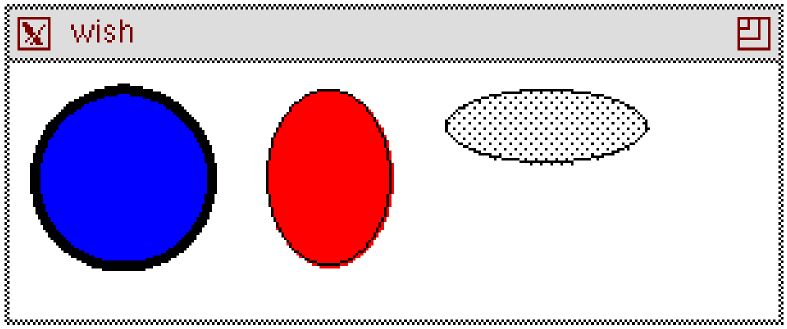
`-arrow` *`where`*	`none first last both`
`-arrowshape` *`{a b c}`*	Three parameters that describe the shape of the arrow. `c` is the width and `b` is the overall length. `a` is the length of the part that touches the line. (e.g., `8 10 3`)
`-capstyle` *`what`*	`butt projecting round`
`-fill` *`color`*	The color of the line.
`-joinstyle` *`what`*	`bevel miter round`
`-smooth` *`boolean`*	If `true`, a spline curve is drawn.
`-splinesteps` *`num`*	Number of line segments that approximate the spline.
`-stipple` *`bitmap`*	Stipple pattern for line fill.
`-tags` *`tagList`*	Set of tags for the line item.
`-width` *`width`*	Width of the line, in screen units.

Oval Items

An `oval` is defined by two sets of coordinates that define its bounding box. If the box is square, a circle is drawn. You can set the color of the interior of the oval as well as the outline of the oval. A sampler of ovals is shown in the next example.

Example 19–9 Canvas `oval` items

```
$c create oval 10 10 80 80 -fill red -width 4
$c create oval 100 10 150 80 -fill blue -width 0
$c create oval 170 10 250 40 -fill black -stipple gray25.
```



The various artifacts on the ovals are a function of the quality of your X server . Different X servers draw circles better and faster than others. Table 19–1 gives the complete set of `oval` attributes.

Table 19–5 Attributes for `oval` canvas items.

`-fill` *`color`*	The color of the interior of the oval.
`-outline` *`color`*	The color for the outline of the oval.
`-stipple` *`bitmap`*	Stipple pattern for oval fill.
`-tags` *`tagList`*	Set of tags for the oval item.
`-width` *`width`*	The thickness of the outline.

Polygon Items

A `polygon` is a closed shape specified by a number of sets of points, one for each vertex of the polygon. The vertices can be connected with smooth or straight lines. There is no outline option for a polygon. You can get an outline by drawing a line with the same coordinates, although you will need to duplicate the starting point at the end of the list of coordinates for the line.

Example 19–10 Canvas `polygon` items.

```
$c create poly 20 -40 40 -20 40 20 20 40 -20 40 \
   -40 20 -40 -20 -20 -40 -fill red
$c create line 20 -40 40 -20 40 20 20 40 -20 40 \
   -40 20 -40 -20 -20 -40 20 -40 -fill white -width 5
$c create text 0 0 -text STOP -fill white
$c move all 50 50
```

Table 19–1 gives the complete set of `polygon` attributes.

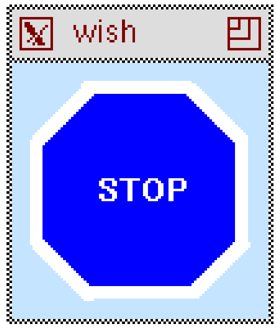


Table 19–6 Attributes for `polygon` canvas items.

`-fill` *`color`*	The color of the polygon.
`-smooth` *`boolean`*	If `true`, a spline curve is drawn around the points.
`-splinesteps` *`num`*	Number of line segments that approximate the spline.
`-stipple` *`bitmap`*	Stipple pattern for polygon fill.
`-tags` *`tagList`*	Set of tags for the line item.

Rectangle Items

A `rectangle` is specified with two coordinates that are its opposite corners. A rectangle can have a fill color and an outline color . The example below drags out a box as the user drags the mouse. All it requires is remembering the last rectangle drawn so it can be delete when the next box is drawn.

Example 19–11 Dragging out a box.

```
proc BoxInit {} {
    canvas .c -bg white ; pack .c
    bind .c <Button-1> {BoxBegin %W %x %y}
    bind .c <B1-Motion> {BoxDrag %W %x %y}
}
proc BoxBegin { w x y } {
    global box
    set box(anchor) [list $x $y]
    catch {unset box(last)}
}
proc BoxDrag { w x y } {
    global box
    catch {$w delete $box(last)}
    set box(last) [eval {$w create rect} $box(anchor) \
        {$x $y -tag box}]
}
```

The example uses `box(anchor)` to record the start of the box. This is a list with two elements. The `eval` command is used so that this list can be spliced into the `create rect` command. T able 19–1 gives the complete set of rectangle attributes.

Table 19–7 Attributes for rectangle canvas items.

`-fill` *color*	The color of the interior of the rectangle.
`-outline` *color*	The color for the outline of the rectangle.
`-stipple` *bitmap*	Stipple pattern for rectangle fill.
`-tags` *tagList*	Set of tags for the rectangle item.
`-width` *width*	The thickness of the outline.

Text Items

The canvas `text` item provides yet another way to display and edit text. It supports selection, editing, and can extend onto multiple lines. The position of a text item is given by one set of coordinates and an anchor position. The size of the text is determined by the number of lines and the length of each line. By default a new line is started if there is a newline in the text string. If a width is specified, in screen units, then any line that is longer than this is wrapped onto multiple lines. The wrap occurs before a space character .

The editing and selection operations for text items operations use indices to specify positions within a given text item. These are very similar to those used in the entry widget. T able 19–8 summarizes the indices for canvas `text` items.

Table 19–8 Indices for canvas `text` items

`0`	Index of the first character .
`end`	Index just past the last character .
`number`	Index a character , counting from zero.
`insert`	The character right after the insertion cursor .
`sel.first`	The first character in the selection.
`sel.last`	The last character in the selection.
`@`*x*`,`*y*	The character under the specified X and Y coordinate.

There are several canvas operations that manipulate text items. These are similar to some of the operations of the entry widget, except that they are parameterized by the tag or ID of the canvas object being manipulated. If the tag refers to more than one object, then the operations apply to the first object in the display list that supports an insert cursor. Table 19–9 summarizes these operations. In the table `$t` is a text item or tag and `$c` is a canvas.

Table 19–9 Canvas operations that apply to `text` items.

`$c dchars $t` *`first ?last?`*	Delete the characters from *`first`* through *`last`*, or just the character at *`first`*.
`$c focus ?$t?`	Set input focus to the specified item, or return the id of the item with the focus if not it is given.
`$c icursor $t` *`index`*	Set the insert cursor to just before *`index`*.
`$c index $t index`	Return the numerical value of *`index`*.
`$c insert $t` *`index string`*	Insert the string just before *`index`*.
`$c select adjust $t` *`index`*	Move the boundary of an existing selection.
`$c select clear`	Clear the selection.
`$c select from $t` *`index`*	Start a selection.
`$c select item`	Returns the id of the selected item, if any.
`$c select to $t` *`index`*	Extend the selection to the specified *`index`*.

There are no default bindings for canvas text items. The following example sets up some rudimentary bindings for canvas text items. The `<Button-1>` and `<Button-2>` bindings are on the canvas as a whole. The rest of the bindings are on items with the `text` tag. The bindings try to be careful about introducing temporary variables because they execute at the global scope. This is a hint that it might be better to create a procedure for each binding.

The `<Button-1>` binding uses the canvas `find overlapping` operation to see if a text object has been clicked. This operation is a little more awkward than the `find closest` operation, but `find closest` will find an object no matter how far away it is.

The `<Button-2>` binding does one of two things. It pastes the selection into the canvas item that has the focus. If no item has the focus, then a new text item is created with the selection as its value.

Example 19–12 Simple edit bindings for canvas text items.

```
proc CanvasEditBind { c } {
    bind $c <Button-1> {
        focus %W
        if {[%W find overlapping [expr %x-2] [expr %y-2] \
                [expr %x+2] [expr %y+2]] == {}} {
            %W focus {}
        }
    }
    $c bind text <Button-1> {
        %W focus current
```

```
        %W icursor current @%x,%y
        %W select from current @%x,%y
    }
    $c bind text <B1-Motion> {
        %W select to current @%x,%y
    }
    $c bind text <Delete> {
        if {[%W select item] != {}} {
            %W dchars [%W select item] sel.first sel.last
        } elseif {[%W focus] != {}} {
            %W dchars [%W focus] insert
        }
    }
    $c bind text <Control-d> {
        if {[%W focus] != {}} {
            %W dchars [%W focus] insert
        }
    }
    $c bind text <Control-h> {
        if {[%W select item] != {}} {
            %W dchars [%W select item] sel.first sel.last
        } elseif {[%W focus] != {}} {
            set _t [%W focus]
            %W icursor $_t [expr [%W index $_t insert]-1]
            %W dchars $_t insert
            unset _t
        }
    }
    $c bind text <BackSpace> [$c bind text <Control-h>]

    $c bind text <Control-Delete> {
        %W delete current
    }
    $c bind text <Return> {
        %W insert current insert \n
    }
    $c bind text <Any-Key> {
        %W insert current insert %A
    }
    bind $c <Button-2> {
        if {[catch {selection get} _s] == 0} {
            if {[%W focus] != {}} {
                %W insert [%W focus] insert $_s
            } else {
                %W create text %x %y -text $_s -anchor nw \
                    -tag text
            }
            unset _s
        }
    }
    $c bind text <Key-Right> {
        %W icursor current [expr [%W index current insert]+1]
    }
    $c bind text <Control-f> [$c bind text <Key-Right>]
    $c bind text <Key-Left> {
```

```
        %W icursor current [expr [%W index current insert]-1]
    }
    $c bind text <Control-b> [$c bind text <Key-Left>]
}
```

Table 19–1 gives the complete set of attributes for `text` items. Note that there are no foreground and background attributes. Instead, the fill color specifies the color for the text. It is possible to stipple the text, too.

Table 19–10 Attributes for `text` canvas items.

`-anchor` *position*	`c n ne e se s sw w nw`
`-fill` *color*	The foreground color for the text.
`-font` *font*	The font for the text.
`-justify` *how*	`left right center`
`-stipple` *bitmap*	Stipple pattern for the text fill.
`-tags` *tagList*	Set of tags for the rectangle item.
`-text` *string*	The string to display.
`-width` *width*	The thickness of the outline.

Window Items

A `window` item allows you to position other Tk widgets on a canvas. The position is specified by one set of coordinates and an anchor position. You can also specify the width and height, or you can let the widget determine its own size. The example below uses a canvas to provide a scrolling surface for a large set of labeled entries. A frame is created and a set of labeled entry widgets are packed into it. This main frame is put onto the canvas as a single window item. This way we let the packer take care of arranging all the labeled entries. The size of the canvas is set up so that a whole number of labeled entries are displayed. The scroll region and scroll increment are set up so that clicking on the scrollbar arrows brings one new labeled entry completely into view.

Example 19–13 Using a `canvas` to scroll a set of widgets.

```
proc Example { top title labels } {
    # Create a resizable toplevel window
    toplevel $top
    wm minsize $top 200 100
    wm title $top $title

    # Create a frame for buttons,
    # Only Dismiss does anything useful
    set f [frame $top.buttons -bd 4]
```

```
    button $f.quit -text Dismiss -command "destroy $top"
    button $f.save -text Save
    button $f.reset -text Reset
    pack $f.quit $f.save $f.reset -side right
    pack $f -side top -fill x

    # Create a scrollable canvas
    frame $top.c
    canvas $top.c.canvas -width 10 -height 10 \
       -yscrollcommand [list $top.c.yscroll set]
    scrollbar $top.c.yscroll -orient vertical \
       -command [list $top.c.canvas yview]
    pack $top.c.yscroll -side right -fill y
    pack $top.c.canvas -side left -fill both -expand true
    pack $top.c -side top -fill both -expand true

    SetOfLabeledEntries $top.c$top.canvas $labels
}
proc SetOfLabeledEntries { canvas labels } {
    # Create one frame to hold everything
    # and position it on the canvas
    set f [frame $canvas.f -bd 0]
    $canvas create window 0 0 -anchor nw -window $f

    # Find out how big the labels are
    set max 0
    foreach label $labels {
       set len [string length $label]
       if {$len > $max} {
          set max $len
       }
    }
    # Create and pack the labeled entries
    set i 0
    foreach label $labels {
       frame $f.$i
       label $f.$i.label -text $label -width $max
       entry $f.$i.entry
       pack $f.$i.label -side left
       pack $f.$i.entry -side right -fill x
       pack $f.$i -side top -fill x
       incr i
    }
    set child [lindex [pack slaves $f] 0]

    # Wait for the window to become visible and then
    # set up the scroll region and increment based on
    # the size of the frame and the subframes

    tkwait visibility $child
    set incr [winfo height $child]
    set width [winfo width $f]
    set height [winfo height $f]
    $canvas config -scrollregion "0 0 $width $height"
    $canvas config -scrollincrement $incr
```

```
    if {$height > 4 * $incr} {
        set height [expr 4 * $incr]
    }
    $canvas config -width $width -height $height
}
Example .top "An example" {
    alpha beta gamma delta epsilon zeta eta theta iota kappa
    lambda mu nu xi omicron pi rho sigma tau upsilon
    phi chi psi omega}
```

The `tkwait visibility` command is important to the example. It causes the script to suspend execution until the toplevel window, `$top`, is displayed on the screen. The wait is necessary so the right information gets returned by the `winfo width` and `winfo height` commands. By waiting for a subframe of the main frame, `$child`, we ensure that the packer has gone through all its processing to position the interior frames. The canvas's scroll region is set to be just large enough to hold the complete frame. The scroll increment is set to the height of one of the subframes.

Canvas Operations

Table 19–11 below summarizes the operations on canvas widgets. In the table, `$t` represents a tag that identifies one or more canvas objects, or it represents the numerical ID of a single canvas object. In some cases an operation only operates on a single object. If a tag identifies several objects, the first object in the display list is operated on.

The canvas display list refers to the global order among canvas objects. New objects are put at the end of the display list. Objects later in the display list obscure objects earlier in the list. The term *above* refers to objects later in the display list.

Table 19–9 describes several of the canvas operations that only apply to `text` objects. They are `dchars focus index icursor insert select`. Those operations are not repeated in the next table. In the table, `$t` is a text item or tag and `$c` is a canvas.

Table 19–11 Operations on a `canvas` widget.

`$c addtag` *`tag`* `above $t`	Add *`tag`* to the item just above `$t` in the display list.
`$c addtag` *`tag`* `all`	Add *`tag`* to all objects in the canvas.
`$c addtag` *`tag`* `below $t`	Add *`tag`* to the item just below `$t` in the display list.

Table 19–11 Operations on a `canvas` widget.

`$c addtag` *`tag`* `closest` *`x y ?halo? ?start?`*	Add *`tag`* to the item closest to the *`x y`* position. If more than one object is the same distance away, or if more than one object is within halo pixels, then the last one in the display list (uppermost) is returned. If *`start`* is specified, the closest object after *`start`* in the display list is returned.
`$c addtag` *`tag`* `enclosed` *`x1 y1 x2 y2`*	Add *`tag`* to the items completely enclosed in the specified region. *`x1`* <= *`x2`*, *`y1`* <= *`y2`*.
`$c addtag` *`tag`* `withtag $t`	Add *`tag`* to the items identified by `$t`.
`$c bbox $t ?`*`tag tag`* `...?`	Return the bounding box of the items identified by the tag(s) in the form *`x1 y1 x2 y2`*
`$c bind $t` *`?sequence? ?command?`*	Set or query the bindings of canvas items.
`$c canvasx` *`screenx ?grid?`*	Map from the X screen coordinate *`screenx`* to the X coordinate in canvas space, rounded to multiples of *`grid`* if specified.
`$c canvasy` *`screeny ?grid?`*	Map from screen Y to canvas Y.
`$c cget` *`option`*	Return the value of *`option`* for the canvas.
`$c configure ...`	Query or update the attributes of the canvas.
`$c coords $t ?`*`x1 y1`* `...?`	Query or modify the coordinates of the item.
`$c create type x y ?x2 y2? ?opt value ...?`	Create a canvas object of the specified *`type`* at the specified coordinates.
`$c delete $t ?tag ...?`	Delete the item(s) specified by the tag(s) or IDs.
`$c dtag $t ?`*`deltag`*`?`	Remove the specified tags from the items identified by `$t`. If *`deltag`* is omitted, it defaults to `$t`
`$c find` *`addtagSearch`* `...`	Return the IDs of the tags that match the search specification: `above all below closest enclosed withtag` as for the `addtag` operation.
`$c gettags $t`	Return the tags associated with the first item identified by `$t`.
`$c itemconfigure $t ...`	Query or reconfigure item $t.
`$c lower $t` *`?belowThis?`*	Move the items identified by `$t` to the beginning of the display list, or just before *`belowThis`*.
`$c move $t dx dy`	Move `$t` by the specified amount.
`$c postscript ...`	Generate postscript. See the next section.
`$c raise $t ?`*`aboveThis`*`?`	Move the items identified by $t to the end of the display list, or just after *`aboveThis`*.

Table 19–11 Operations on a `canvas` widget.

`$c scale $t` *`x0 y0 xS yS`*	Scale the coordinates of the items identified by $t. The distance between *`x0`* and a given X coordinate changes by a factor of *`xS`*. Similarly for Y.
`$c scan mark` *`x y`*	Set a mark for a scrolling operation.
`$c scan dragto x y`	Scroll the canvas from the previous mark.
`$c type $t`	Return the type of the first item identified by `$t`.
`$c xview` *`index`*	Position the canvas so that *`index`* (in scroll increments) is at the left edge of the screen.
`$c yview` *`index`*	Position the canvas so that *`index`* (in scroll increments) is at the top edge of the screen.

Generating Postscript

The `postscript` operation generates postscript based on the contents of a canvas. There are many options that refine the postscript. You control what region of the canvas is printed with the `-width`, `-height`, `-x` and `-y` options. You control the size and location of this in the output with the `-pageanchor`, `-pagex`, `-pagey`, `-pagewidth`, and `-pageheight` options. The postscript is written to the file named by the `-file` option, or it is returned as the value of the postscript canvas operation.

You control fonts with a mapping from X screen fonts to postscript fonts. Define an array where the index is the name of the X font and the contents are the name and pointsize of a postscript font.

Table 19–12 summarizes all the options for generating postscript.

Table 19–12 Canvas postscript options.

`-colormap varName`	The index of varName is a named color, and the contents of each element is the postscript code to generate the RGB values for that color.
`-colormode` *`mode`*	*`mode`* is one of: `color grey mono`
`-file` *`name`*	The file in which to write the postscript. If not specified, the postscript is returned as the result of the command.
`-fontmap` *`varName`*	The index of varName is an X font name. Each element contains a list of two items, a postscript font name and a point size.
`-height` *`size`*	Height of the area to print.
`-pageanchor` *`anchor`*	*`anchor`* is one of: `c n ne e se s sw s nw`

Table 19–12 Canvas postscript options.

`-pageheight` *size*	Height of image on the output. A floating point number followed by `c` (centimeters) `i` (inches) `m` (millimeters) `p` (printer points).
`-pagewidth` *size*	Width of image on the output.
`-pagex` *position*	The output X coordinate of the anchor point.
`-pagey` *position*	The output Y coordinate of the anchor point.
`-rotate` *boolean*	If true, rotate so that X axis is the long direction of the page (landscape orientation).
`-width` *size*	Size of the area to print.
`-x` *position*	Canvas X coordinate of left edge of the image.
`-y` *position*	Canvas Y coordinate of top edge of the image.

The next example positions a number of text objects with different fonts onto a canvas. For each different X font used, it records a mapping to a postscript font. The example has a fairly simple font mapping, and in fact the canvas would have probably guessed the same font mapping itself. If you use more exotic screen fonts you may need to help the canvas widget with an explicit font map.

The example positions the output at the upper-left corner of the printed page by using the `-pagex`, `-pagey` and `-pageanchor` options. Recall that postscript has its origin at the lower left corner of the page.

Example 19–14 Generating postscript from a `canvas`.

```
proc Setup {} {
    global fontMap
    canvas .c
    pack .c -fill both -expand true
    set x 10
    set y 10
    set last [.c create text $x $y -text "Font sampler" \
        -font fixed -anchor nw]

    # Create several strings in different fonts and sizes

    foreach family {times courier helvetica} {
        set weight bold
        switch -- $family {
            times { set fill blue; set psfont Times}
            courier { set fill green; set psfont Courier }
            helvetica { set fill red; set psfont Helvetica }
        }
        foreach size {10 14 24} {
            set y [expr 4+[lindex [.c bbox $last] 3]]

            # Guard against missing fonts
```

```
            if {[catch {.c create text $x $y \
                    -text $family-$weight-$size \
                    -anchor nw -fill $fill \
                    -font -*-$family-$weight-*-*-*-$size-*} \
            it] == 0} {
                set fontMap(-*-$family-$weight-*-*-*-$size-*)\
                    [list $psfont $size]
                set last $it
            }
        }
    }
    set fontMap(fixed) [list Courier 12]
}
proc Postscript { c file } {
    global fontMap
    # Tweak the output color
    set colorMap(blue) {0.1 0.1 0.9 setrgbcolor}
    set colorMap(green) {0.0 0.9 0.1 setrgbcolor}
    # Position the text at the upper-left corner of
    # an 8.5 by 11 inch sheet of paper
    $c postscript -fontmap fontMap -colormap colorMap \
        -file $file \
        -pagex 0.i -pagey 11.i -pageanchor nw
}
```

Canvas Attributes

Table 19–13 lists the attributes for the `canvas` widget. The table uses the X resource Class name, which has capitals at internal word boundaries. In Tcl commands the attributes are specified with a dash and all lowercase.

Table 19–13 Resource names of attributes for the `canvas` widget.

`background`	The normal background color.
`borderWidth`	The width of the border around the canvas.
`closeEnough`	Distance from mouse to an overlapping object.
`confine`	Boolean. True constrains view to the scroll region.
`cursor`	Cursor to display when mouse is over the widget.
`height`	Height, in screen units, of canvas display.
`highlightColor`	Color for input focus highlight border.
`highlightThickness`	Width of highlight border.
`insertBackground`	Background for area covered by insert cursor.
`insertBorderwidth`	Width of cursor border. Non-zero for 3D effect.
`insertOffTime`	Time, in milliseconds the insert cursor blinks off.

Table 19–13 Resource names of attributes for the `canvas` widget.

`insertOnTime`	Time, in milliseconds the insert cursor blinks on.
`insertWidth`	Width of insert cursor. Default is 2.
`relief`	3D relief: `flat, sunken, raised, groove, ridge`.
`scrollIncrement`	The minimum scrolling distance.
`scrollRegion`	Left, top, right, and bottom coordinates of `the canvas`
`selectBackground`	Background color of selection.
`selectForeground`	Foreground color of selection.
`selectBorderWidth`	Widget of selection border. Non-zero for 3D effect.
`width`	Width, in characters for text, or screen units for image.
`xScrollCommand`	Tcl command prefix for horizontal scrolling.
`yScrollCommand`	Tcl command prefix for vertical scrolling.

The scroll region of a canvas defines the boundaries of the canvas coordinate space. It is specified as four coordinates, *`x1 y1 x2 y2`* where (*`x1, y1`*) is the top-left corner and (*`x2, y2`*) is the lower right corner. If the `constrain` attribute is true, then the canvas cannot be scrolled outside this region. It is OK to position canvas objects partially or totally off the scroll region, they just may not be visible. The scroll increment determines how much the canvas is scrolled when the user clicks on the arrows in the scroll bar.

The `closeEnough` attribute indicates how far away a position can be from an object and still be considered to overlap it. This applies to the `overlapping` search criteria.

Hints

Large coordinate spaces

Coordinates for canvas items are stored internally as floating point numbers, so the values returned by the `coords` operation will be floating point numbers. If you have a very large canvas, you may need to adjust the precision with which you see coordinates by setting the `tcl_precision` variable. This is an issue if you query coordinates, perform a computation on them, and then update the coordinates.

Scaling and Rotation

The scale operation scales the coordinates of one or more canvas items. It is not possible to scale the whole coordinate space. The main problem with this is that you can lose precision when scaling and unscaling objects because their internal coordinates are actually changed by the scale operation. For simple cases this is not a problem, but in extreme cases it can show up.

The canvas does not support rotation.

X Resources

There is no resource database support built into the canvas and its items. You can, however, define resources and query them yourself. For example, you could define

```
*Canvas.foreground:blue
```

This would have no effect by default. However, your code could look for this resource with `option get`. You'd then have to specify this color directly for the `-fill` attribute of your objects.

```
set fg [option get $c foreground {}]
$c create rect 0 0 10 10 -fill $fg
```

The main reason to take this approach is to let your users customize the appearance of canvas objects without changing your code.

Objects with many points

The canvas implementation seems well optimized to handle lots and lots of canvas objects. However, if an object like a line or a polygon has very many points that define it, the implementation ends up scanning through these points linearly. This can adversely affect the time it takes to process mouse events in the area of the canvas containing such an item. Apparently any object in the vicinity of a mouse click is scanned to see if the mouse has hit it so that any bindings can be fired.

CHAPTER 20

Selections and the Clipboard

Cut and paste allows information exchange between applications. The X selection mechanism is used for this purpose. The clipboard selection is a special-purpose selection mechanism used by the OpenLook tools.

Selections are handled in a general way by X, including a provision for different selections, different data types, and different formats for the data. For the most part you can ignore these details because they are handled by the Tk widgets. However, you can also control the selection explicitly. This chapter describes how.

There are two Tcl commands that deal with selections. The `selection` command is a general purpose command that can set and get different selections. By default it manipulates the `PRIMARY` selection. The `clipboard` command is used to store data for later retrieval using `CLIPBOARD` selection. The next example implements a robust paste operation by checking for both of these selections.

Example 20–1 Paste the `PRIMARY` or `CLIPBOARD` selection.

```
proc Paste { t } {
    if [catch {selection get} sel] {
        if [catch {selection get -selection CLIPBOARD} sel] {
            # no selection or clipboard data
            return
        }
    }
    $t insert insert $sel
}
```

The selection Command

The basic model for selections is that there is an owner for a selection, and other applications request the value of the selection from that owner. The X server keeps track of ownership, and applications are informed when some other application takes away ownership. Several of the Tk widgets implement selections and take care of asserting ownership and returning its value. The `selection get` command returns the value of the current selection, or raises an error if the selection does not exist. The error conditions are checked in the previous example.

For many purposes the selection handling that is built into the Tk widgets is adequate. If you want more control over selection ownership, you can provide a handler for selection requests. The last section of this chapter presents an example of this.

A selection can have a type. The default is `STRING`. The type is different than the name of the selection (e.g., `PRIMARY` or `CLIPBOARD`). Each type can have a format, and the default is also `STRING`. Ordinarily these defaults are fine. If you are dealing with non-Tk applications, however, you may have to ask for their selections by the right type (e.g., `FILE_NAME`). Other formats include `ATOM` and `INTEGER`. An `ATOM` is a name that is registered with the X server and identified by number. It is probably not a good idea to use non-`STRING` types and formats because it limits what other applications can use the information. The details about selection types and formats are specified in the *Inter-Client Communication Conventions Manual* (ICCCM).

Table 20–1 summarizes the `selection` command. All of the operations take an optional parameter that specifies what selection is being manipulated. This defaults to `PRIMARY`. Some of the operations take a pair of parameters that specify what X display the selection is on. The value for this is a Tk pathname of a window, and the selection on that window's display is manipulated. The default is to manipulate the selection on the display of the main window.

Table 20–1 The `selection` command.

`selection clear ?-displayof` *`win`*`? ?-selection` *`sel`*`?`	Clear the specified selection.
`selection get ?displayof` *`win`*`? ?-selection` *`sel`*`? ?-type` *`type`*`?`	Return the specified selection. Type defaults to `STRING`.
`selection handle ?-selection` *`sel`*`? ?-type` *`type`*`? ?-format` *`format`*`?` *`window command`*	Define *`command`* to be the handler for selection requests when *`window`* owns the selection.
`selection own ?-displayof` *`window`*`? ?-selection` *`sel`*`?`	Return the Tk pathname of the window that owns the selection, if it is in this application.
`selection own ?-command` *`command`*`? ?-selection` *`sel`*`?` *`window`*	Assert that *`window`* owns the *`sel`* selection. The *`command`* is called when ownership of the selection is taken away from *`window`*.

The clipboard Command

The `clipboard` command is used to install values into the `CLIPBOARD` selection. The `CLIPBOARD` is meant for values that have been recently or temporarily deleted. The `selection` command is used to retrieve values from the `CLIPBOARD` selection. For example, the `Paste` function in Example 20–1 will insert from the `CLIPBOARD` if there is no `PRIMARY` selection. Table 20–1 summarizes the `clipboard` command.

Table 20–2 The `clipboard` command.

`clipboard clear ?-displayof` *`win`*`?`	Clear the `CLIPBOARD` selection.
`clipboard append ?-displayof` *`win`*`? ?-format` *`format`*`? ?-type` *`type`*`?` *`data`*	Append *`data`* to the `CLIPBOARD` with the specified *`type`* and *`format`*, which both default to `STRING`.

Interoperation with OpenLook

The `CLIPBOARD` is necessary to interoperate correctly with OpenLook. When the user presses the `Copy` or `Cut` function keys in an OpenLook application, a value is copied into the `CLIPBOARD`. A `Paste` inserts the contents of the `CLIPBOARD`; the contents of the `PRIMARY` selection are ignored.

In contrast, toolkits like Tk and Xt that use the `PRIMARY` selection do not need a `Copy` step. Instead, dragging out a selection with the mouse automatically asserts ownership of the `PRIMARY` selection, and paste inserts the value of the `PRIMARY` selection.

Selection Handlers

The `selection handle` command registers a Tcl command to handle selection requests. The command is called to return the value of the selection to a requesting application. If the selection value is large, the command may be called several times to return the selection in pieces. The command gets two parameters that indicate the offset within the selection to start returning data, and the maximum number of bytes to return. If the command returns fewer than that many bytes, the selection request is assumed to be completed. Otherwise the command is called again to get the rest of the data, and the offset parameter is adjusted accordingly.

You can also get a callback when you lose ownership of the selection. At that time it is appropriate to unhighlight the selected object in your interface. The `selection own` command is used to set up ownership and register a callback when you lose ownership.

A canvas selection handler

The following example illustrates a selection handler for a canvas widget. A description of the selected object is returned in such a way that the requester can create an identical object. The example lacks highlighting for the selected object, but otherwise provides full cut, copy and paste functionality .

Example 20–2 A selection handler for `canvas` widgets.

```
proc SetupCanvasSelect { c } {
    # Create a canvas with a couple of objects
    canvas $c
    pack $c
    $c create rect 10 10 50 50 -fill red -tag object
    $c create poly 100 100 100 30 140 50 -fill orange \
        -tag object
    # Set up cut and paste bindings
    $c bind object <1> [list CanvasSelect $c %x %y]
    $c bind object <3> [list CanvasCut $c %x %y]
    bind $c <2> [list CanvasPaste $c %x %y]
    # Register the handler for selection requests
    selection handle $c [list CanvasSelectHandle $c]
}

proc CanvasSelect { w x y } {
    # Select an item on the canvas.
    # This should highlight the object somehow, but doesn't
    global canvas
    set id [$w find closest $x $y]
    set canvas(select,$w) $id
    # Claim ownership of the PRIMARY selection
    selection own -command [list CanvasSelectLose $w] $w
}

proc CanvasCut { w x y } {
    # Delete an object from the canvas, saving its
    # description into the CLIPBOARD selection
    global canvas
    set id [$w find closest $x $y]
    # Clear the selection so Paste gets the clipboard
    selection clear
    clipboard clear
    clipboard append [CanvasDescription $w $id]
    $w delete $id
}

proc CanvasSelectHandle { w offset maxbytes } {
    # Handle a selection request
    global canvas
    if ![info exists canvas(select,$w)] {
        error "No selected item"
    }
    set id $canvas(select,$w)
    # Return the requested chunk of data.
```

```
	return [string range [CanvasDescription $w $id] \
		$offset [expr $offset+$maxbytes]]
}
proc CanvasDescription { w id } {
	# Generate a description of the object that can
	# be used to recreate it later.
	set type [$w type $id]
	set coords [$w coords $id]
	set config {}
	# Bundle up non-default configuration settings
	foreach conf [$w itemconfigure $id] {
		# itemconfigure returns a list like
		# -fill {} {} {} red
		set default [lindex $conf 3]
		set value [lindex $conf 4]
		if {[string compare $default $value] != 0} {
			append config [list [lindex $conf 0] $value] " "
		}
	}
	return [concat CanvasObject $type $coords $config]
}

proc CanvasSelectLose { w } {
	# Some other app has claimed the selection
	global canvas
	unset canvas(select,$w)
}

proc CanvasPaste { w x y } {
	# Paste the selection, from either the
	# PRIMARY or CLIPBOARD selections
	if [catch {selection get} sel] {
		if [catch {selection get -selection CLIPBOARD} sel] {
			# no selection or clipboard data
			return
		}
	}
	if [regexp {^CanvasObject} $sel] {
		if [catch {eval {$w create} [lrange $sel 1 end]} id] {
			return;
		}
		# look at the first coordinate to see where to
		# move the object. Element 1 is the type, the
		# next two are the first coordinate
		set x1 [lindex $sel 2]
		set y1 [lindex $sel 3]
		$w move $id [expr $x-$x1] [expr $y-$y1]
	}
}
```

CHAPTER 21

Callbacks and Handlers

This chapter describes the `send` command that is used to invoke Tcl commands in other applications, the `after` command that causes Tcl commands to occur at a time in the future, and the `fileevent` command that registers a command to occur in response to file I/O.

Callbacks and interprocess communication provide powerful mechanisms for structuring your application. The `send` command lets Tk applications send each other Tcl commands and cooperate in very flexible ways. A large application can be structured as a set of smaller tools that cooperate instead of one large monolith. This encourages reuse, and it exploits your workstations multiprogramming capabilities. Within a single application you can use the `after` command to cause events to occur at a specified time in the future. This is useful for periodic tasks and animations. The `fileevent` command lets you application do I/O processing in the background and response as needed when I/O events occur. Together, all of these mechanisms support a flexible and powerful applications.

The after Command

The `after` command sets up commands to happen in the future. In its simplest form it just pauses the application for a specified time, in milliseconds. During this time the application processes no events. This behavior is different than the `tkwait` command that does allow event processing. The example below waits for half a second.

```
after 500
```

The `after` command can register a Tcl command to occur after period of time, in milliseconds. The `after` command behaves like `eval`; if you give it extra arguments it concatenates them to form a single command. If your argument structure is important, use `list` to build the command. The following example always works, no matter what the value of `myvariable` is.

```
after 500 [list puts $myvariable]
```

The return value of `after` is an identifier for the registered command. You can cancel this command with the `after cancel` operation. You specify either the identifier returned from `after`, or the command string. In the latter case the event that matches the command string exactly is canceled.

Table 21–1 summarizes the `after` command.

Table 21–1 The `after` command.

`after` *`milliseconds`*	Pause for *`milliseconds`*.
`after` *`ms arg`* `?`*`arg`*`...?`	Concatenate the *`args`* into a command and execute it after *`ms`* milliseconds. Immediately returns an ID.
`after cancel` *`id`*	Cancel the command registered under *`id`*.
`after cancel` *`command`*	Cancel the registered *`command`*.

The fileevent Command

The `fileevent` command registers a procedure that is called when an I/O stream is ready for read or write events. For example, you can open a pipeline for reading, and then process the data from the pipeline using a command registered with `fileevent`. The advantage of this approach is that your application can do other things, like update the user interface, while waiting for data from the pipeline. If you use a Tcl extension like Tcl-DP that lets you open network I/O streams, then you can also use `fileevent` to register procedures to handle data from those I/O streams. You can use `fileevent` on `stdin` and `stdout`, too.

The command registered with `fileevent` uses the regular Tcl commands to read or write data on the I/O stream. For example, if the pipeline generates line-oriented output, you can use `gets` to read a line of input. If you try and read more data than is available, your application will hang waiting for more input. For this reason you should read one line in your fileevent handler, assuming the data is line-oriented. If you know the pipeline will generate data in fixed-sized blocks, then you can use the `read` command to read one block.

Currently there is no support for non-blocking writes, so there is a chance that writing too much data on a writable I/O stream will block your process.

You should check for end-of-file in your read handler because it will be called when end-of-file occurs. It is safe to close the stream inside the file handler. Closing the stream automatically unregisters the handler.

There can be at most one read handler and one write handler for an I/O stream. If you register a handler and one is already registered, then the old registration is removed. If you call `fileevent` without a command argument it returns the currently registered command, or null if there is none. If you register the null string, it deletes the current file handler.

The example below shows a typical read event handler. A pipeline is opened for reading and its command executes in the background. The `Reader` command is invoked when data is available on the pipe. The end-of-file condition is checked, and then a single line of input is read and processed. Example 11–1 also uses `fileevent` to read from a pipeline.

Example 21–1 A read event file handler.

```
set pipe [open "|some command"]
fileevent $pipe readable [list Reader $pipe]
proc Reader { pipe } {
    if [eof $pipe] {
        catch {close $pipe}
        return
    }
    gets $pipe line
    # Process one line
}
```

Table 21–1 summarizes the `fileevent` command.

Table 21–2 The `fileevent` command.

`fileevent` *`fileId`* `readable ?`*`command`*`?`	Query or register *`command`* to be called when *`fileId`* is readable.
`fileevent` *`fileId`* `writable ?`*`command`*`?`	Query or register *`command`* to be called when *`fileId`* is writable.

The send Command

The `send` command invokes a Tcl command in another application. This provides a very general way for scripts to cooperate. The general form of the command is

```
send interp arg ?arg...?
```

Perhaps the trickiest thing to get right with `send` is *`interp`*, which is the name of the other application. An application defines its own name when it creates its main window. The *wish* shell uses as its name the last component of the filename of the script it is executing. For example, if the file `/usr/local/bin/exmh` begins with:

```
#!/usr/local/bin/wish
```

The *wish* shell will interpret the script and set up its application name to be `exmh`. However, if another instance of the `exmh` application is already running, then *wish* will choose the name `exmh #2`, and so on. If *wish* is not executing from a file, then its name is just `wish`. You may have noticed `wish #2` or `wish #3` in your window title bars, and this reflects the fact that multiple *wish* applications are running on your display. If you application crashes it can forget to unregister its name. The *tkinspect* program has a facility to clean up these old registrations.

A script can find out its own name, so you can pass names around or put them into files in order to set up communications. The tk appname command is used to get or change the application name.

```
set myname [tk appname]
tk appname aNewName
```

In Tk 3.6 and earlier, you have to use the `winfo name` command to get the name of the application.

```
set myname [winfo name .]
```

The sender script

The following example is a general purpose script that reads input and then sends it to another application. You can put this at the end of a pipeline in order to get a loopback effect to the main application, although you can also use `fileevent` for similar effects. One advantage of `send` over `fileevent` is that the sender and receiver can be more independent. A logging application, for example, can come and go independently of the applications that log error messages.

Example 21–2 The sender application.

```
#!/usr/local/bin/wish
# sender takes up to four arguments:
# 1) the name of the application to which to send.
# 2) a command prefix
# 3) the name of another application to notify when
# after the end of the data.
# 4) the command to use in the notification.

# Hide the unneeded window
wm withdraw .

# Process command line arguments
if {$argc == 0} {
   puts stderr "Usage: send name ?cmd? ?uiName?"
   exit 1
} else {
   set app [lindex $argv 0]
}
if {$argc > 1} {
   set cmd [lindex $argv 1]
} else {
   set cmd Send_Insert
}
if {$argc > 2} {
   set ui [lindex $argv 2]
   set uiCmd Send_Done
}
```

```
if {$argc > 3} {
    set uiCmd [lindex $argv 3]
}
# Read input and send it to the logger
while {! [eof stdin]} {
    set numBytes [gets stdin input]
    if {$numBytes < 0} {
        break
    }
    # Ignore errors with the logger
    catch {send $app [concat $cmd [list $input\n]]}
}
# Notify the controller, if any
if [info exists ui] {
    if [catch {send $ui $uiCmd} msg] {
        puts stderr "send.tcl could not notify $ui\n$msg"
    }
}
# This is necessary to force wish to exit.
exit
```

The *sender* application supports communication with two processes. It sends all its input to a primary "logging" application. When the input finishes, it can send a notification message to another "controller" application. The logger and the controller could be the same application. An example that sets up this three way relationship is given later.

Consider the `send` command used in the example:

```
send $app [concat $cmd [list $input\n]]
```

The combination of `concat` and `list` is a little tricky. The `list` command is used to quote the value of the input line. This quoted value is then appended to the command so it appears as a single extra argument. Without the quoting by `list`, the value of the input line will affect the way the remote interpreter parses the command. Consider these alternatives:

```
send $app [list $cmd $input]
```

This form is safe, except that it limits `$cmd` to be a single word. If `cmd` contains a value like the ones given below, the remote interpreter will not parse it correctly. It will treat the whole multi-word value as the name of a command.

```
.log insert end
.log see end ; .log insert end
```

The version below is the most common wrong answer:

```
send $app $cmd $input
```

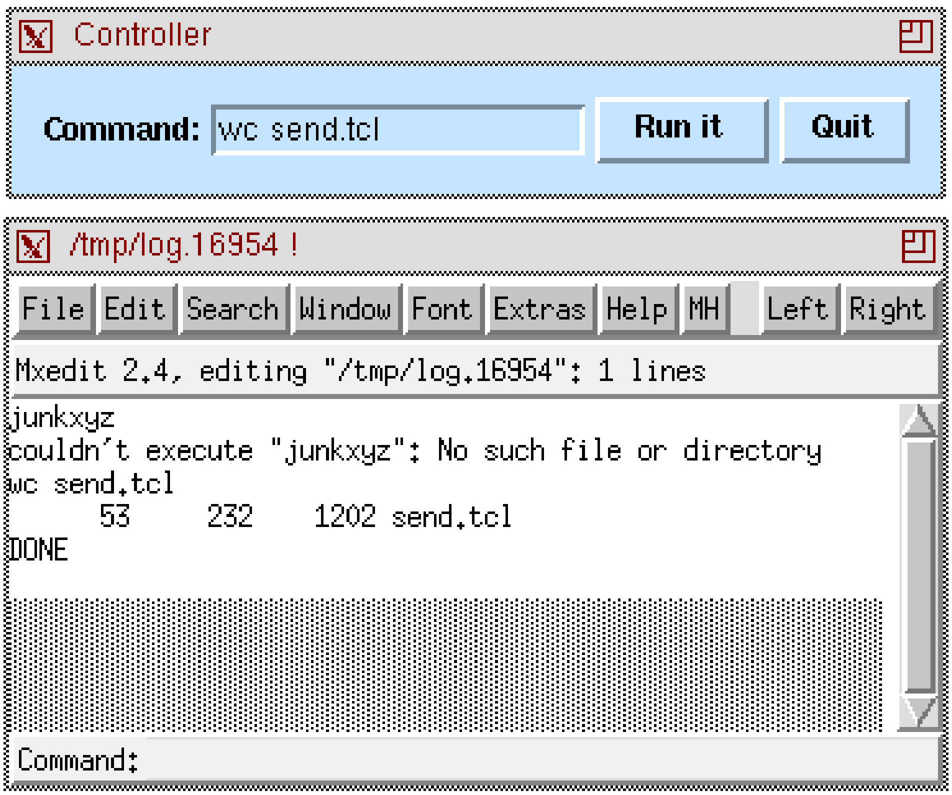
The `send` command will concatenate `$cmd` and `$input` together, and the result will be parsed again by the remote interpreter. The success or failure of the remote command depends on the value of the input data, which is always a bad idea.

Using sender

The example below is taken from a control panel that runs jobs in the background and uses sender to send their output to an editor for logging. When the job finishes, the control panel is notified.

The editor is *mxedit*, a Tcl-based editor. It defines its application name to be `mxedit` *`pathname`*, where pathname is the name of the file being edited. That name will be passed to sender as the name of the logging application. The control panel passes its own name as the name of the controller, and it uses the `tk appname` command to find out its own name.

Example 21–3 Using the sender application.




```
#!/project/tcl/src/brent/wish
# Send chapter
# Control Panel demo

wm title . Controller

# Create a frame for buttons and entry.
frame .top -borderwidth 10
pack .top -side top -fill x
```

```
# Create the command buttons.
button .top.quit -text Quit -command exit
set but [button .top.run -text "Run it" -command Run \
    -width 6]
pack .top.quit .top.run -side right

# Create a labeled entry for the command
label .top.l -text Command: -padx 0
entry .top.cmd -width 20 -relief sunken \
    -textvariable command
pack .top.l -side left
pack .top.cmd -side left -fill x -expand true

# Set up key binding equivalents to the buttons
bind .top.cmd <Return> Run
bind .top.cmd <Control-c> Stop
focus .top.cmd

# Fork an editor to log the output.
exec mxedit /tmp/log.[pid] &

set sendCmd [list /usr/local/bin/send.tcl \
        "mxedit /tmp/log.[pid]" mxInsert [tk appname]]

# Run the program and arrange to log its input via sender
proc Run {} {
    global command job sendCmd but
    set cmd [concat exec $command |& $sendCmd &]
    send "mxedit /tmp/log.[pid]" [list mxInsert $command\n]
    if [catch {eval $cmd} job] {
        send "mxedit /tmp/log.[pid]" [list mxInsert $job\n]
    } else {
        $but config -text Stop -command Stop
    }
}
# Stop the program and fix up the button
proc Stop {} {
    global job but
    # job contains multiple pids
    catch {eval {exec kill} $job}
    send "mxedit /tmp/log.[pid]" [list mxInsert ABORT\n]
    $but config -text "Run it" -command Run
}
# Handle the callback from sender
proc Send_Done {} {
    global but
    send "mxedit /tmp/log.[pid]" [list mxInsert DONE\n]
    $but config -text "Run it" -command Run
}
```

This example is very similar to the `ExecLog` application from Example 11–1 on page 104. Instead of creating a text widget for a log, this version forks the *mxedit* program to serve as the logging application. The command is run in a pipeline. Instead of reading the pipeline itself, the control panel lets the sender program send the output to the editor. When the process completes, the sender notifies the control panel.

The formation of the command to execute is done in two parts. First, the `sendCmd` variable is set up with the right arguments to send.tcl. This includes the result of `tk appname`, which gives the name of the controller application. Once again, it is crucial to use `list` so that spaces in the names of the interpreters are quoted properly. In the second step the user's command is concatenated into a pipeline command, and `eval` is used to interpret the carefully constructed command.

The return from `exec` is a list of process ids, one for each process in the pipeline. This leads to another use of `eval` to construct a `kill` command that lists each process id as separate arguments.

The example always uses `list` to construct the command used in a `send`. In this case it is necessary in order to preserve the newline character that is appended to the string being inserted. Another approach would be to use curly braces. In that case the `\n` would be converted to a newline character by the remote interpreter. However, this doesn't work when the command or error message is being sent. In these cases the variable needs to be expanded, so `list` is used in all cases for consistency.

Hooking the browser to a shell

Chapter 11 presented two examples, a browser for the examples in this book and a simple shell in which to try out Tcl commands. The two examples shown below hook these two applications together using the `send` command. The first example adds a `Load` button to the browser that tells the shell to source the current file. The browser starts up the shell, if necesarry.

Example 21–4 Hooking the browser to an `eval` server.

```
# Add this to Example 11-2
button .menubar.load -text Load -command Load
pack .menubar.load -side right

# Start up the eval.tcl script.
proc StartEvalServer {} {
	global browse
	# Start the shell and pass it our name.
	exec eval.tcl [tk appname] &
	# Wait for eval.tcl to send us its name
	tkwait variable browse(evalInterp)
}
proc Load {} {
	global browse
	if {[lsearch [winfo interps] eval.tcl] < 0} {
		StartEvalServer
	}
	if [catch {send $browse(evalInterp) {info vars}} err] {
```

```
        # It probably died - restart it.
        StartEvalServer
    }
    # Send the command, using after 1 so that _EvalServe
    # is done asynchronously. We don't wait. The three
    # list commands foil the concat done by send, after, and
    # the uplevel in _EvalServe
    send $browse(evalInterp) \
        [list after 1 [list _EvalServe \
            [list source $browse(current)]]]
}
```

The number of lists created before the send command may seem excessive. Here is what happens. First, the `send` command concats its arguments, so instead of letting it do that, we pass it a single list. The `after` command also concats its arguments, so it is passed a list as well. If you didn't care how long the command would take, you could eliminate the use of `after` to simplify things. Finally, `_EvalServe` expects a single argument that is a valid command, so `list` is used to construct that.

We need to add two things to Example 11–3 to get it to support these additions to the browser. First, when it starts up it needs to send us its application name. We pass our own name on its command line, so it knows how to talk to us. Second, an `_EvalServer` procedure is added. It accepts a remote command, echos it in the text widget, and then evaluates it. The results, or errors, are added to the text widget.

Example 21–5 Making the shell into an `eval` server.

```
# Add this to the beginning of Example 11–3
if {$argc > 0} {
    # Check in with the browser
    send [lindex $argv 0] \
        [list set browse(evalInterp) [tk appname]]
}

# Add this after _Eval
proc _EvalServe { command } {
    global prompt

    set t .eval.t
    $t insert insert $command\n

    set err [catch {uplevel #0 $command} result]
    $t insert insert \n$result\n
    $t insert insert $prompt
    $t see insert
    $t mark set limit insert
}
```

CHAPTER 22

Tk Widget Attributes

Each Tk widget has a number of attributes that affect its appearance and behavior. This chapter describes the use of attributes in general, and covers some of the size and appearance-related attributes. The next two chapters cover the attributes associated with colors, images, and text.

This chapter describes some of the widget attributes that are in common among many Tk widgets. A widget always provides a default value for its attributes, so you can avoid specifying most of them. If you want to fine tune things, however, you'll need to know about all the widget attributes. You may want to just skim through this chapter first, and then refer back to it once you have learned more about a particular Tk widget.

Configuring Attributes

Attributes for Tk widgets are specified when the widget is created. They can be changed dynamically at any time after that, too. In both cases the syntax is similar, using pairs of arguments. The first item in the pair identifies the attribute, the second provides the value. For example, a button can be created like this:

```
button .doit -text Doit -command DoSomething
```

The name of the button is `.doit`, and two attributes are specified, the `text` and the `command`. The `.doit` button can be changed later with the `configure` widget operation:

```
.doit configure -text Stop -command StopIt
```

The current configuration of a widget can be queried with another form of the `configure` operation. If you just supply an attribute, the settings associated with that attribute are returned:

```
.doit configure -text
=> -text text Text { } Stop
```

This command returns several pieces of information: the command line switch, the resource name, the resource class, the default value, and the current value. In most cases you want the current value, which comes last. One way to get this value is with `lindex`:

```
lindex [.doit configure -text] 4
```

Tk 4.0 added a `cget` widget command that makes life easier. Just do:

```
.doit cget -text
=> Stop
```

You can also configure widget attributes indirectly by using the X resource database. An advantage of using the resource database is that users can reconfigure your application without touching the code. Otherwise, if you specify attribute values explicitly in the code, they cannot be overridden by resource settings. This is especially important for attributes like fonts and colors.

The tables in this chapter list the attributes by their X resource name, which may have a capital letter at an internal word boundary (e.g., `activeBackground`). When you specify attributes in a Tcl command, use all lowercase instead, plus a leading dash. Compare:

```
option add *Button.activeBackground red
$button configure -activebackground red
```

The first command defines a resource that affects all buttons created after that point, while the second command changes an existing button. Command line settings override resource database specifications. Chapter 27 describes the use of X resources in more detail.

Size

Most widgets have a `width` and `height` attribute that specifies their desired size, although there are some special cases described below. In all cases, the geometry manager for a widget may modify the size to some degree. The table below summarizes the attributes used to specify the size for widgets.

Table 22–1 Size attribute resource names.

`aspect`	The aspect ratio of a message widget, which is 100 times the ratio of width divided by height. `message`
`height`	Height, in text lines or screen units. `button canvas checkbutton frame label listbox listbox menubutton radiobutton text toplevel`
`length`	The long dimension of a scale. `scale`

Table 22–1 Size attribute resource names.

`orient`	Orientation for long and narrow widgets: `horizontal vertical.` `scale scrollbar.`
`width`	Width, in characters or screen units. `button canvas checkbutton entry frame label listbox menubutton message radiobutton scale scrollbar text toplevel`

Most of the text-related widgets interpret their sizes in units of characters for width and lines for height. All other widgets, including the `message` widget, interpret their dimensions in screen units. Screen units are pixels by default, although you can suffix the dimension with a unit specifier:

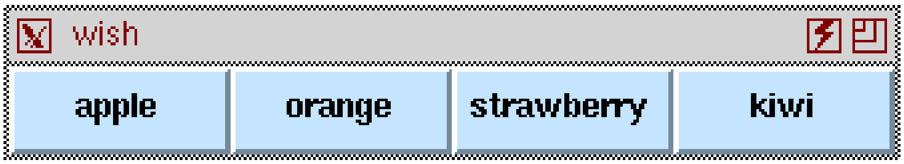
```
c    centimeters
i    inch
m    millimeters
p    printer points (1/72 inches)
```

Scales and scrollbars can have two orientations as specified by the `orient` attribute, so width and height are somewhat ambiguous. These widgets do not support a `height` attribute, and they interpret their `width` attribute to mean the size of their narrow dimension. The `scale` has a `length` attribute that determines its long dimension. Scrollbars do not even have a `length`. Instead, a `scrollbar` is assumed to be packed next to the widget it controls, and the `fill` packing attribute is used to extend the scrollbar to match the length of its adjacent widget. Example 15–5 shows how to pack scrollbars with another widget.

The `message` widget displays a fixed string on multiple lines, and it uses one of two attributes to constrain its size: its `aspect` or its `width`. The aspect ratio is defined to be 100*width/height, and it formats its text to honor this constraint. However, if a `width` is specified, it just uses that and uses as many lines (i.e. as much height) as needed. Example 15–2 and Example 15–3 show how message widgets display text.

It is somewhat unfortunate that text-oriented widgets only take character- and line-oriented dimensions. These sizes change with the font used for the label, and if you want a precise size you might be frustrated. One trick is to put each widget, such as a label, in its own frame. Specify the size you want for the frame, and then pack the label and turn off size propagation. For example:

Example 22–1 Equal-sized labels




```
proc EqualSizedLabels { parent width height strings args } {
    set l 0
    foreach s $strings {
        frame $parent.$l -width $width -height $height
        pack propagate $parent.$l false
        pack $parent.$l -side left
        eval {label $parent.$l.l -text $s} $args
        pack $parent.$l.l -fill both -expand true
        incr l
    }
}
frame .f ; pack .f
EqualSizedLabels .f 1i 1c {apple orange strawberry kiwi} \
    -relief raised
```

The frames `$parent.$l` are all created with the same size. The `pack propagate` command prevents these frames from changing size when the labels are packed into them later. The labels are packed with `fill` and `expand` turned on so they fill up the fixed-sized frames around them.

Borders and Relief

The three dimensional appearance of widgets is determined by two attributes: `borderWidth` and `relief`. The `borderWidth` adds extra space around the edge of a widget's display, and this area can be displayed in a number of ways according to the `relief` attribute. The example on the next page illustrates the different reliefs

Table 22–2 Border and relief attribute resource names.

`borderWidth`	The width of the border around a widget, in screen units. `button canvas checkbutton entry frame label listbox menu menubutton message radiobutton scale scrollbar text toplevel`
`bd`	`Short for borderWidth. Tcl commands only.`

Table 22–2 Border and relief attribute resource names.

`relief`	The appearance of the border:
	`flat raised sunken ridge groove`
	`button canvas checkbutton entry frame label`
	`listbox menubutton message radiobutton scale`
	`scrollbar text toplevel`

`activeBorderWidth`	The borderwidth for `menu` entries.
`activeRelief`	The relief for a active `scrollbar` elements.

Example 22–2 3D relief sampler.


```
frame .f -borderwidth 10
pack .f
foreach relief {raised sunken flat ridge groove} {
   label .f.$relief -text $relief -relief $relief -bd 4
   pack .f.$relief -side left -padx 4
}
```

The `activeBorderWidth` attribute is a special case for menus. It defines the border width for the menu entries. The relief of a menu is (currently) not configurable. It probably isn’t worth adjusting the menu border width attributes because the default looks OK.

The `activeRelief` applies to the elements of a scrollbar (the elevator and two arrows) when the mouse is over them. In this case there is no corresponding border width to play with, and changing the `activeRelief` doesn’t look that great.

The Focus Highlight

Each widget can have a focus highlight that indicates what widget currently has the input focus. This is a thin rectangle around each widget that is displayed in the normal background color by default. When the widget gets the input focus, the highlight rectangle is displayed in an alternate color . The addition of the highlight adds a small amount of space outside the border described in the previous section. Attributes control the width and color of this rectangle. If the widget is zero, no highlight is displayed.

By default, only the widgets that normally expect input focus have a non-

Table 22–3 Border and relief attribute resource names.

`highlightColor`	The color of the highlight when the widget has focus.
`highlightThickness`	The width of the highlight border .

zero width highlight border . This includes the `text`, `entry`, and `listbox` widgets. It also includes the `button` and `menu` widgets because there is a set of keyboard traversal bindings that focus input on these widgets, too.

Padding and Anchors

Some widgets have padding and anchor attributes that are similar in spirit to some packing attributes described in Chapter 12, *The Pack Geometry Manager* . However , they are distinct from the packing attributes, and this section explains how they work together with the packer

Table 22–4 Layout attribute resource names.

`anchor`	The anchor position of the widget: `n ne e se s sw w nw center`. `button, checkbutton, label, menubutton, message, radiobutton`.
`padX, padY`	Padding space in the X or Y direction, in screen units. `button checkbutton label menubutton message radiobut-ton text`

The padding attributes for a widget define space that is never occupied by the display of the widgets contents. For example, if you create a `label` with the following attributes and pack it into a frame by itself, you will see the text is still centered, in spite of the `anchor` attribute.


```
label .foo -text Foo -padx 20 -anchor e
pack .foo
```

The `anchor` attribute only affects the display if there is extra room for another reason. One way to get extra room is to specify a `width` attribute that is longer than the text. The following label has right-justified text. You can also see the effect of the default `padx` value for labels that keeps the text spaced away from the right edge.

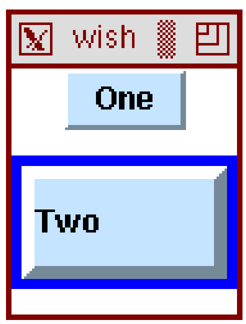
```
label .foo -text Foo -width 10 -anchor e
pack .foo
```

Another way to get extra display space is with the `-ipadx` and `-ipady` packing parameters. The example in the next section illustrates this effect. Chapter 12 has several more packing examples of the packing parameters.

Putting It All Together

The number of different attributes that contribute to the size and appearance can be confusing. The example in this section uses a label to demonstrate the difference among size, borders, padding, and the highlight. Padding can come from the geometry manager , and it can come from widget attributes.

Example 22–3 Borders and padding.




```
frame .f -bg white
label .f.one -text One -relief raised
pack .f.one -side top
label .f.two -text Two \
    -highlightthickness 4 -highlightcolor red \
    -borderwidth 5 -relief raised \
    -padx 0 -pady 0 \
    -width 10 -anchor w
pack .f.two -side top -pady 10 -ipady 10 -fill both
focus .f.two
pack .f
```

The first `label` in the example uses a raised relief so you can see the default 2-pixel border . There is no highlight on a label by default. There is one pixel of internal padding so that the text is spaced away from the edge of the label. The second label adds a highlight rectangle by specifying a non-zero thickness. W id- gets like buttons, entries, listboxes, and text have a highlight rectangle by default. The second label's padding attributes are reduced to zero. The anchor positions the text right next to the border in the upper left (`nw`) corner. However, note the effect of the padding provided by the packer. There is both external and internal padding in the Y direction. The external padding (from `pack -pady`) results in unfilled space. The internal packing (`pack -ipady`) is used by the label for its display. This is different than the label's own `-pady` attribute, which keeps the text away from the edge of the widget.

CHAPTER 23

Color, Images, and Cursors

This chapter describes the color attributes shared by the Tk widgets. Images and bitmaps can be displayed instead of text by several widgets. This chapter describes that commands that create and manipulate images. The shape of the mouse cursor when it is over a particular widget is also controlled by attributes. This chapter includes a figure that shows all the cursors in the X cursor font.

Color is one of the most fun things to play with in a user interface. However, this chapter makes no attempt to improve your taste in color choices; it just describes the attributes that affect color. Because color choices are often personal, it is a good idea to specify them via X resources so your users can change them easily. For example, Tk does not have a reverse video mode. However, with a couple resource specifications you can convert a monochrome display into reverse video. The definitions are given in the next example. The `Foreground` and `Background` class names are used, and the various foreground and background colors (e.g., `activeBackground`) are given the right resource class so these settings work out.

Example 23–1 Resources for reverse video.

```
proc ReverseVideo {} {
    option add *Foreground white
    option add *Background black
}
```

This chapter describes images, too. The image facility in Tk lets you create an image and then have other Tk widgets display it. The same image can be displayed by many different widgets (or multiple times on a canvas). If you redefine an image, its display is updated in whatever widgets are displaying it.

The last topic of the chapter is cursors. All widgets can control what the mouse cursor looks like when it is over them. In addition, the widgets that support text input define another cursor, the insert cursor. Its appearance is controlled with a few related attributes.

Colors

There are several color attributes. The `foreground` color is used to draw an element, while the `background` color is used for the blank area behind the element. Text, for example, is painted with the foreground color. There are several variations on foreground and background that reflect different states for widgets or items they are displaying. Table 23–1 lists the resource names for color attributes. The table indicates what widgets use the different color attributes. Remember to use all lowercase and a leading dash when specifying attributes in a Tcl command.

Table 23–1 Color attribute resource names.

`background`	The normal background color. `button canvas checkbutton entry frame label listbox menu menubutton message radiobutton scale scrollbar text toplevel`
`bg`	Short for background. Command line only.
`foreground`	The normal foreground color. `button checkbutton entry label listbox menu menubutton message radiobutton scale text`
`fg`	Short for foreground. Command line only.
`activeBackground`	The background when a mouse button will take an action. `button checkbutton menu menubutton radiobutton scale scrollbar`
`activeForeground`	The foreground when the mouse is over an active widget. `button checkbutton menu menubutton radiobutton`
`disabledForeground`	The foreground when a widget is disabled. `button checkbutton menu menubutton radiobutton`
`highlightColor`	The color for input focus highlight. `button canvas checkbutton entry frame label menubutton radiobutton scale scrollbar text toplevel`
`insertBackground`	The background for the area covered by the insert cursor. `canvas entry text`
`selectBackground`	The background of selected items. `canvas entry listbox text`
`selectColor`	The color of the selector indicator. `checkbutton radiobutton`

Table 23–1 Color attribute resource names.

`selectForeground`	The foreground of selected items. `canvas entry listbox text`
`troughColor`	The trough part of scales and scrollbars. `scale scrollbar`

Color values are specified in two ways: symbolically (e.g., `red`), or by hexadecimal numbers (e.g., `#ff0000`). The leading `#` distinguishes the hexadecimal representation from the symbolic one. The number is divided into three equal sized fields that give the red, green, and blue values, respectively. The fields can specify 4, 8, 12, or 16 bits of a color:

`#`*RGB* `4 bits per color`

`#`*RRGGBB* `8 bits per color`

`#`*RRRGGGBBB* `12 bits per color`

`#`*RRRRGGGGBBBB* `16 bits per color`

If you specify more resolution than is supported by the X server, the low order bits of each field are discarded. The different display types supported by X are described in the next section. Each field ranges from 0, which means no color, to a maximum, which is all ones in binary, or all `f` in hex, that means full color saturation. For example, pure red can be specified four ways:

`#f00 #ff0000 #fff000000 #ffff00000000`

The symbolic color names understood by the X server may vary from system to system. You can hunt around for a file named `rgb.txt` in the X directory structure to find a listing of them. Or, run the *xcolors* program that comes with the standard X distribution.

The `winfo rgb` command maps from a color name (or value) to three numbers that are its red, green, and blue values. You can use this to compute variations on a color. The `ColorDarken` procedure shown below uses the `winfo rgb` command to get the red, green, and blue components of the input color. It reduces these amounts by 5 percent, and reconstructs the color specification using the `format` command.

Example 23–2 Computing a darker color

```
proc ColorDarken { color } {
    set rgb [winfo rgb $color]
    return [format "#%03x%03x%03x" \
        [expr round([lindex $rgb 0] * 0.95)] \
        [expr round([lindex $rgb 1] * 0.95)] \
        [expr round([lindex $rgb 2] * 0.95)]]
}
```

Colormaps and Visuals

For the most part Tk manages the color resources of the display for you. However, if your application uses a lot of colors you may need to control the display with the `Visual` and `Colormap` attributes described in this section. Competition from other applications can cause color allocations to fail, and this causes Tk to switch into monochrome mode (i.e., black and white).

Each pixel on the screen is represented by one or more bits of memory. There are a number of ways to map from a value stored at a pixel to the color that appears on the screen at that pixel. The mapping is a function of the number of bits at each pixel, which is called the *depth* of the display, and the style of interpretation, or *visual class*. The 6 visual classes defined by X are listed in the table below. Some of the visuals use a *colormap* that maps from the value stored at a pixel to a value used by the hardware to generate a color. A colormap enables a compact encoding for a much richer color. For example, a 256 entry colormap can be indexed with 8 bits, but it may contain 24 bits of color information. If you run the UNIX *xdpyinfo* program it will report the different visual classes supported by your display.

Table 23–2 Visual classes for X displays. Values for the visual attribute.

`staticgrey`	Greyscale with a fixed colormap defined by the X server.
`greyscale`	Greyscale with a writable colormap.
`staticcolor`	Uses a colormap defined by the X server.
`pseudocolor`	Color values determined by single writable colormap.
`directcolor`	Color values determined by three independent colormaps.
`truecolor`	Color values determined by read-only independent colormaps?

The `frame` and `toplevel` widgets support a `Colormap` and `Visual` attribute that gives you control over these features of the X display. Again, in a Tcl command specify these attributes in all lowercase with a leading dash. Unlike other attributes, these cannot be changed after the widget is created. The value of the Visual attribute has two parts, a visual type and the desired depth of the display. The following example requests a greyscale visual with a depth of 4 bits per pixel.

```
toplevel .grey -visual "greyscale 4"
```

By default a widget inherits the colormap and visual from its parent widget. The value of the `Colormap` attribute can be the keyword `new`, in which case the frame or toplevel gets a new private colormap, or it can be the name of another widget, in which case the frame or toplevel shares the colormap of that widget. When sharing colormaps, the other widget must be on the same screen and using the same visual class.

Bitmaps and Images

The label and all the button widgets have an `image` attribute that specifies a graphic image to display. Using an image takes two steps. In the first step the image is created via the `image create` command. This command returns an identifier for the image, and it is this identifier that is passed to widgets as the value of their image attribute.

Example 23–3 Specifying an image attribute for a widget.

```
set im [image create bitmap \
   -file glyph.bitmap -maskfile glyph.mask \
   -background white -foreground blue]
button .foo -image $im
```

There are three things that can be displayed by labels and all the buttons: text, bitmaps, and images. If more than one of these attributes are specified, then the image has priority over the bitmap, and the bitmap has priority over the text. You can remove the image or bitmap attribute by specifying a null string for its value. The text, if any, will be displayed instead.

The image Command

Table 23–3 summarizes the `image` command.

Table 23–3 Summary of the `image` command.

`image create` *`type`* *`?name? ?options?`*	Create an image of the specified type. If name is not specified, one is made up. The remaining arguments depend on the type of image being created.
`image delete` *`name`*	Delete the named image.
`image height` *`name`*	Return the height of the image, in pixels.
`image names`	Return the list of defined images.
`image type` *`name`*	Return the type of the named image.
`image types`	Return the list of possible image types.
`image width` *`name`*	Return the width of the image, in pixels.

The exact set of options for `image create` depend on the image type. There are two built-in image types: `bitmap` and `photo`. Chapter 30 describes the C interface for defining new image types.

bimap images

A `bitmap` image has a main image and a mask image. The main image is drawn in the foreground color. The mask image is drawn in the background color, unless the corresponding bit is set in the main image. The remaining bits are "clear" and the widget's normal background color shows through. For the `bitmap` image type supports the following options.

Table 23–4 Bitmap image options

`-background` *`color`*	The background color. (*no -bg equivalent*)
`-data` *`string`*	The contents of the bitmap as a string.
`-file` *`name`*	The name of the file containing a bitmap definition.
`-foreground` *`color`*	The foreground color. (*no -fg equivalent*)
`-maskdata` *`string`*	The contents of the mask as a string.
`-maskfile` *`name`*	The name of the file containing the mask data.

The bitmap definition files are stylized C structure definitions that are parsed by X. These are generated by bitmap editors such as `bitmap` program, which comes with the standard X distribution. The `-file` and `-maskfile` options name a file that contains such a definition. The `-data` and `-maskdata` options specify a string in the same format as the contents of one of those files.

The bitmap attribute

The label and all the button widgets also support a `bitmap` attribute, which is a special case of an image. This attribute is a little more convenient than the image attribute because the extra step of creating an image is not required. However, there is some power and flexibility with the `image` command, such as the ability to reconfigure a named image (e.g., for animation) that is not possible with a bitmap. .

Example 23–4 Specifying a bitmap for a widget.

```
button .foo -bitmap @glyph.bitmap -fg blue
```

The `@` syntax for the bitmap attribute signals that a file containing the bitmap is being specified. It is also possible to name built-in bitmaps. The predefined bitmaps are shown in the next figure along with their symbolic name. Chapter X describes the C interface for defining built in bitmaps.

Example 23–5 The built-in bitmaps

```
foreach name {error gray25 gray50 hourglass \
              info questhead question warning} {
   frame .$name
   label .$name.l -text $name -width 9 -anchor w
```

```
    label .$name.b -bitmap $name
    pack .$name.l -side left
    pack .$name.b -side top
    pack .$name -side top -expand true -fill x
}
```

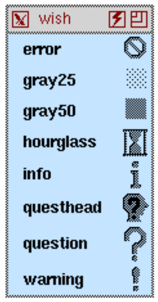


photo images

The `photo` image type was contributed by Paul Mackerras. It displays full color images and can do dithering and gamma correction. The `photo` image supports different image formats, although the only format supported by Tk 4.0 is the PPM format. There is a C interface to define new photo formats.

Table 23–5 lists the attributes for photo images. These are specified in the `image create photo` command.

Table 23–5 Photo image attributes

`-format` *`format`*	Specifies the data format for the file or data string.
`-data` *`string`*	The contents of the photo as a string.
`-file` *`name`*	The name of the file containing a photo definition.
`-gamma` *`value`*	A gamma correction factor, which must be greater than zero. A value greater than one brightens an image.
`-height` *`value`*	The height, in screen units.
`-palette` *`spec`*	A single number specifies the number of grey levels. Three numbers separated by slashes determines the number of red, blue, and green levels.
`-width` *`value`*	The width of the image, in screen units.

The `format` indicates what format the data is in. However, the photo implementation will try all format handlers until it find one that accepts the data. An explicit format limits what handlers are tried. The format name is treated as a prefix that is compared against the names of handlers. Case is not significant in the format name.

The `palette` setting determines how many colors or graylevels are used with rendering an image. If a single number is specified, the image is rendered in greyscale with that many different graylevels. For full color, three numbers separated by slashes specify the number of shades of red, green, and blue, respectively. The more levels you specify the more room you take up in your colormap. The photo widget will switch to a private colormap if necessary. Multiply the number of red, green, and blue shades to determine how many different colors you use. If you have an 8-bit display, there are only 256 colors available. Reasonable palette settings that don't hog the colormap include `5/5/4` and `6/6/5`. You can get away with fewer shades of blue because the human eye is less sensitive to blue.

After you create an image you can operate on it with several image instance operations. In the table below, `$p` is a photo image handle returned by the `image create photo` command.

Table 23–6 Photo image operations.

`$p blank`	Clear the image. It becomes transparent.
`$p cget` *`option`*	Return the configuration attribute *`option`*.
`$p configure ...`	Reconfigure the photo image attributes.
`$p copy` *`source options`*	Copy another image. Table 23–7 lists the copy options.
`$p get` *`x y`*	Return the pixel value at position *`x y`*.
`$p put` *`data`* `?-to` *`x1 y1 x2 y2`*`?`	Insert *`data`* into the image. *`data`* is a list of rows, where each row is a list of colors.
`$p read` *`file options`*	Load an image from a file. Table X lists the read options.
`$p redither`	Reapply the dithering algorithm to the image.
`$p write file options`	Save the image to *`file`* accoring to *`options`*. Table X lists the write options.

Table 23–7 lists the options available when you copy data from one image to another. The regions involved in the copy are specified by the upper-left and lower-right corners. If the lower-right corner of the source is not specified, then it defaults to the lower-right corner of the image. If the lower-right corner of the destination is not specified, then the size is determined by the are of the source. Otherwise, the source image may be cropped or replicated to fill up the destination.

Table 23–7 lists the read options, and Table 23–7 lists the write options. The format option is more important for writing, because otherwise the first format found will be used. With reading, the format is determined automatically, although if there are multiple image types that can read the same data, you can use the format to chose the one you want.

Table 23–7 Image copy options.

`-from` *x1 y1 ?x2 y2?*	Specifies the location and area in the source image. If *x2* and *y2* are not given, there are set to the bottom-right corner.
`-to` *x1 y1 ?x2 y2?*	Specifies the location and area in the destination. If *x2* and *y2* are not given, the size is determined by the source. The source may be cropped or tiled to fill the destination.
`-shrink`	Shrink the destination so its bottom right corner matches the bottom right corner of the data copied in. This has no effect if the `width` and `height` have been set for the image.
`-zoom` *x ?y?*	Magnify the source so each source pixel becomes a block of *x* by *y* pixels. *y* defaults to *x* if it isn't specified.
`-decimate` *x ?y?*	Reduce the source by taking every *x*'th pixel in the X direction and every *y*'th pixel in the Y direction. *y* defaults to *x*.

Table 23–8 Image read options.

`-format` *format*	Specifies the format of the data. By default, the format is determined automatically.
`-from` *x1 y1 ?x2 y2?*	Specifies a subregion of the source data. If *x2* and *y2* are not given, the size is determined by the data.
`-to` *x1 y1*	Specifies the top-left corner of the new data.
`-shrink`	Shrink the destination so its bottom right corner matches the bottom right corner of the data read in. This has no efect if the width and height have been set for the image.

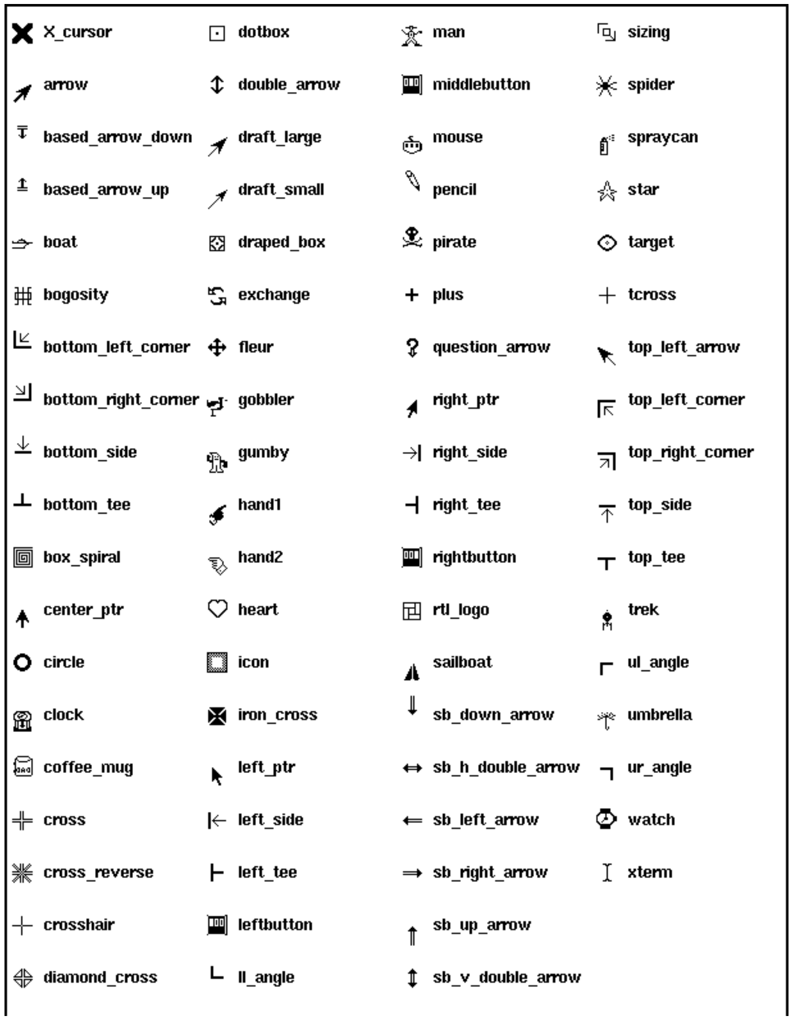
Table 23–9 Image write options.

`-format` *format*	Specifies the format of the data. By default, the format is determined automatically.
`-from` *x1 y1 ?x2 y2?*	Specifies a subregion of the data to save. If *x2* and *y2* are not given, they are set to the lower-right corner.

The Mouse Cursor

The `cursor` attribute defines the mouse cursor. This attribute can have a number of forms. The simplest is a symbolic name for one of the glyphs in the X cursor font, which is shown in the figure on the next page. Optionally, a foreground and background color for the cursor can be specified. Here are some examples:

```
$w config -cursor watch;# stop-watch cursor
$w config -cursor {gumby blue};# blue gumby
```




```
$w config -cursor {X_cursor red white};# red X on white
```

The other form for the cursor attribute specifies a file that contains the definition of the cursor bitmap. If two file names are specified, then the second specifies the cursor mask that determines what bits of the background get covered up. Bitmap editing programs like *idraw* and *iconedit* can be used to generate these files. Here are some example cursor specification using files. You need to specify a foreground color, and if you specify a mask file then you also need to specify a background color.

```
$w config -cursor "@timer.bitmap black"
$w config -cursor "@timer.bitmap timer.mask black red"
```

The Text Insert Cursor

The `text`, `entry`, and `canvas` widgets have a second cursor to mark the text insertion point. The text insert cursor is described by a set of attributes. These attributes can make the insert cursor vary from a thin vertical line to a large rectangle with its own relief. Table 23–10 lists these attributes. The default insert cursor is a 2-pixel wide vertical line. You may not like the look of a wide insert cursor. The cursor is centered between two characters, so a wide one does not look the same as the block cursors found in many terminal emulators. Instead of occupying the space of a single character, it partially overlaps the two characters on either side.

Table 23–10 Cursor attribute resource names.

`cursor`	The mouse cursor. See text for sample formats. `button canvas checkbutton entry frame label listbox menu menubutton message radiobutton scale scrollbar text toplevel`
`insertBackground`	Color for the text insert cursor. canvas entry text
`insertBorderWidth`	Width for three dimensional appearance. canvas entry text
`insertOffTime`	Milliseconds the cursor blinks off. `canvas entry text`
`insertOnTime`	Milliseconds the cursor blinks on. `canvas entry text`
`insertWidth`	Width of the text insert cursor, in screen units. `canvas entry text`

CHAPTER 24

Fonts and Text Attributes

This chapter describes the naming convention for X fonts. The examples show how to trap errors from missing fonts. This chapter describes other text-related attributes such as justification, anchoring, and geometry gridding.

Fonts can cause trouble because the set of installed fonts can vary from system to system. This chapter describes the font naming convention and the pattern matching done on font names. If you use many different fonts in your application, you should specify them in the most general way so the chances of the font name matching an installed font is increased.

After fonts are described, the chapter explains a few of the widget attributes that relate to fonts. This includes justification, anchors, and geometry gridding.

Fonts

Fonts are specified with X font names. The font names are specified with the `-font` attribute when creating or reconfiguring a widget.

```
label .foo -text "Foo" -font fixed
```

This label command creates a label widget with the `fixed` font. `fixed` is an example of a short font name. Other short names might include `6x12`, `9x15`, or `times12`. However , these aliases are site dependent. In fact, all font names are site dependent because different fonts may be installed on different systems. The only font guaranteed to exist is named `fixed`.

The more general form of a font name has several components that describe various attributes of the font. Each component is separated by a dash, and asterisk (*) is used for unspecified components. Short font names are just aliases for these more complete specifications. Here is an example:

```
-*-times-medium-r-normal-*-18-*-*-*-*-*-iso8859-1
```

The components of font names are listed in Table 24–1 in the order in which they occur in the font specification. The table gives the possible values for the components. If there is an ellipsis (...) then there are more possibilities, too.

Table 24–1 X Font specification components.

Component	Description
foundry	`adobe xerox linotype misc ...`
family	`times helvetica lucida courier symbol ...`
weight	`bold medium demibold demi normal book light`
slant	`i r o`
swidth	`normal sans narrow semicondensed`
adstyle	`sans`
pixels	`8 10 12 14 18 24 36 48 72 144 ...`
points	`0 80 100 120 140 180 240 360 480 720 ...`
resx	`0 72 75 100`
resy	`0 72 75 100`
space	`p m c`
avgWidth	`73 94 124 ...`
registry	`iso8859 xerox dec adobe jisx0208.1983 ...`
encoding	`1 fontspecific dectech symbol dingbats`

The most common attributes chosen for a font are its family, weight, slant, and size. The family determines the basic look, such as `courier` or helvetica. The weight is usually **bold** or medium. The slant component is a bit cryptic, but `i` means *italic*, `r` means roman (i.e., normal), and `o` means *oblique*. A given font family might have an italic version, or an oblique version, but not both. Similarly, not all weights are offered by all font families. Size can be specified in pixels (i.e., screen pixels) or points. Points are meant to be independent of the screen resolution. On a 75dpi font, there are about 10 points per pixel. Again, not all font sizes are available in all fonts.

It is generally a good idea to specify just a few key aspects of the font and use `*` for the remaining components. The X server will attempt to match the font specification with its set of installed fonts, but it will fail if there is a specific component that it cannot match. If the first or last character of the font name is an asterisk, then that can match multiple components. The following selects a 12 pixel times font:

```
*times-medium-r-*-*-12*
```

Two useful UNIX programs that deal with X fonts are *xlsfonts* and *xfontsel*. These are part of the standard X11 distribution. `xlsfonts` simply lists the available fonts that match a given font name. It uses the same pattern matching that the server does. Because asterisk is special to most UNIX shells, you'll need to quote the font name argument if you run `xslfonts` from your shell. `xfontsel` has a graphical user interface and displays the font that matches a given font name.

Unfortunately, if a font is missing, neither Tk nor the X server attempt to substitute another font, not even `fixed`. The `FindFont` routine looks around for an existing font. It falls back to `fixed` if nothing else matches.

Example 24–1 FindFont matches an existing font.

```
proc FindFont { w {sizes 14} {weight medium} {slant r}} {
    foreach family {times courier helvetica} {
        foreach size $sizes {
            if {[catch {$w config -font \
                -*-$family-$weight-$slant-*-*-$size-*}] == 0} {
                return -*-$family-$weight-$slant-*-*-$size-*
            }
        }
    }
    $w config -font fixed
    return fixed
}
```

The `FindFont` proc takes the name of a widget, `w`, as an argument, plus some optional font characteristics. All five kinds of text widgets take a `-font` attribute specification, so you can use this routine on any of them. The `sizes` argument is a set of pixel sizes for the font (not points). The routine is written so you can supply a choice of sizes, but it fixes the set of families it uses and allows only a single weight and slant. Another approach is to loop through a set of more explicit font names, with `fixed` being your last choice. The font that works is returned by the procedure so that the search results can be saved and reused later. This is important because opening a font for the first time is a fairly heavyweight operation, and a failed font lookup is also expensive.

Another approach to the font problem is to create a wrapper around the Tk widget creation routines. While you are at it you can switch some attributes to positional arguments if you find you are always specifying them.

Example 24–2 Handling missing font errors.

```
proc Button { name text command args } {
    set cmd [list button $name -text $text -command $command]
    if [catch {concat $cmd $args} w] {
        puts stderr "Button (warning) $w"
```

```
        # Delete the font specified in args, if any
        set ix [lsearch $args -font]
        if {$ix >= 0} {
            set args [lreplace $args $ix [expr $ix+1]]
        }
        # This font overrides the resource database
        eval $cmd $args {-font fixed}
    }
    return $name
}
```

The `Button` procedure creates a button and always takes a `text` and `command` argument. Note that `list` is used to carefully construct the prefix of the Tcl command so that the values of `text` and `command` are preserved. Other arguments are passed through with `args`. The procedure falls back to the `fixed` font if the button command fails. It is careful to eliminate the font specified in `args`, if it exists. The explicit font overrides any setting from the resource database or the Tk defaults. Of course, it might fail for some more legitimate reason, but that is allowed to happen in the backup case. The next example provides a generate wrapper that can be used when creating any widget.

Example 24–3 FontWidget protects against font errors.

```
proc FontWidget { args } {
    if [catch $args w] {
        # Delete the font specified in args, if any
        set ix [lsearch $args -font]
        if {$ix >= 0} {
            set args [lreplace $args $ix [expr $ix+1]]
        }
        # This font overrides the resource database
        set w [eval $args {-font fixed}]
    }
    return $w
}
FontWidget button .foo -text Foo -font garbage
```

Text Layout

There are two simple text layout attributes, `justify` and `wrapLength`. The `text` widget introduces several more layout-related attributes, and Chapter X describe those in detail. The two attributes described in this section apply to the various `button` widgets, the `label`, `entry`, and `message` widgets. Those widgets are described in Chapters 14, 15, and 16.

The `justify` attribute causes text to be centered, left-justified, or right justified. The default justification is `center` for all the widgets in the table, except for the `entry` widget that is left-justified by default.

The `wrapLength` attribute specifies how long a line of text is before it is

Table 24–2 Resource names for layout attributes.

`justify`	Text line justification: `left center right`. `button checkbutton entry label menubutton message radiobutton`
`wrapLength`	Maximum line length for text, in screen units. `button checkbutton label menubutton radiobutton`

wrapped onto another line. It is used to create multi-line buttons and labels. This attribute is specified in screen units, however , not string length. It is probably easier to achieve the desired line breaks by inserting newlines into the text for the button or label and specifying a `wrapLength` of 0, which is the default.

Padding and Anchors

Some widgets have padding and anchor attributes that are similar in spirit to some packing attributes described in Chapter 12, *The Pack Geometry Manager* . However , they are distinct from the packing attributes, and this section explains how they work together with the packer

Table 24–3 Resource names for padding and anchors.

`anchor`	The anchor position of the widget: `n ne e se s sw w nw center`. `button, checkbutton, label, menubutton, message, radiobutton`.
`padX, padY`	Padding space in the X or Y direction, in screen units. `button checkbutton label menubutton message radiobut-ton text`

The padding attributes for a widget define space that is never occupied by the display of the widgets contents. For example, if you create a `label` with the following attributes and pack it into a frame by itself, you will see the text is still centered, in spite of the `anchor` attribute.


```
label .foo -text Foo -padx 20 -anchor e
pack .foo
```

The `anchor` attribute only affects the display if there is extra room for another reason. One way to get extra room is to specify a `width` attribute that is longer than the text. The following label has right-justified text.

```
label .foo -text Foo -width 10 -anchor e
```


```
pack .foo
```

Another way to get extra display space is with the `-ipadx` and `-ipady` packing parameters. The following commands produce approximately the same display as the last example. With the packing parameters you specify things in screen dimensions instead of string lengths.

```
label .foo -text Foo -anchor e
pack .foo -ipadx 25
```

Chapter 12 has several more packing examples that illustrate the effects of the packing parameters.

Gridding, Resizing, and Geometry

The `text` and `listbox` widgets support geometry gridding. This is an alternate interpretation of the main window geometry that is in terms of grid units, typically characters, as opposed to pixels. The `setGrid` attribute is a boolean that indicates if gridding should be turn on. The widget implementation takes care of defining a grid size that matches its character size.

When a widget is gridded, its size is constrained to have a whole number of grid units displayed. In other words, the height will be constrained to show a whole number of text lines, and the width will be constrained to show a whole number of average width characters. This affects interactive resizing by users, as well as the various window manger commands (`wm`) that relate to geometry. When gridding is turned on, the geometry argument (e.g., `24x80`) is interpreted as grid units, otherwise it is interpreted as pixels. The window manager geometry commands are summarized below. In all cases, the `win` parameter to the `wm` command is a toplevel window. However, widget that asks for gridding is typically an interior window surrounded by a collection of other widgets.

Table 24–4 Geometry commands affected by gridding.

`wm geometry` *`win`* `?`*`geometry`*`?`	Set or query the geometry of a window.
`wm minsize` *`win`* `?`*`width height`*`?`	Set the minimum window size.
`wm maxsize` *`win`* `?`*`width height`*`?`	Set the maximum window size.
`wm grid win ?`*`width height dw dh`*`?`	Define the grid parameters.

An important side-effect of gridding is that it enables interactive resizing by the user. Setting the minsize or maxsize of a window also enables resizing.

Otherwise, Tk windows are only resizable under program control. Try out the following example with and without the `-setgrid` flag., and with and without the `wm minsize` command. The `ScrolledListbox` procedure is defined on page 183.

Example 24–4 A gridded, resizable listbox.

```
wm minsize . 20 20
button .quit -text Quit -command exit
pack .quit -side top -anchor e
frame .f
pack .f -side top -fill both -expand true
ScrolledListbox .f -width 10 -height 5 -setgrid true
```

Selection Attributes

Each widget can export its selection via the X selection mechanism. This is controlled with the `exportSelection` attribute. The colors for selected text are set with `selectForeground` and `selectBackground` attributes. The selection is drawn in a raised relief, and the `selectBorderWidth` attribute affects the 3D appearance. Choose a border width of zero to get a flat relief.

A Font Selection Application

This chapter concludes with an application that lets you browse the fonts available in your system. This is modeled after the *xfontsel* program. It displays a set of menus, one for each component of a font name. You can select different values for the components, although the complete space of font possibilities is not defined. You might choose components that result in an invalid font name. The tool also lets you browse the list of available fonts, though, so you can find out what is offered.

Example 24–5 A font selection application.

```
#!/import/tcl/bin/wish+
# The menus are big, so position the window
# near the upper-left corner of the display
wm geometry . +30+30

# Create a frame and buttons along the top
frame .buttons
pack .buttons -side top -fill x
button .buttons.quit -text Quit -command exit
button .buttons.reset -text Reset -command Reset
pack .buttons.quit .buttons.reset -side right

# An entry widget is used for status messages
entry .buttons.e -textvar status -relief flat
```

```
pack .buttons.e -side top -fill x
proc Status { string } {
   global status
   set status $string
   update idletasks
}
# So we can see status messages
tkwait visibility .buttons.e
```

The application uses the font global variable for its state. It creates a status line and a few buttons at the top. Underneath that is a set of menus, one for each font component. The next example creates the menu buttons for the menus.

Example 24–6 Menus for each font component.

```
# Set up the menus. There is one for each
# component of a font name, except that the two resolutions
# are combined and the avgWidth is suppressed.
frame .menubar
set font(comps) {foundry family weight slant swidth \
   adstyle pixels points res res2 \
   space avgWidth registry encoding}
foreach x $font(comps) {
   # font(component) lists all possible component values
   # font(cur,component) keeps the current component values
   set font(cur,$x) *
   set font($x) {}
   # Trim out the second resolution and the average width
   if {$x == "res2" || $x == "avgWidth"} {
      continue
   }
   # The border and highlight thickness are set to 0 so the
   # button texts run together into one long string.
   menubutton .menubar.$x -menu .menubar.$x.m -text -$x \
      -padx 0 -bd 0 -font fixed \
      -highlightthickness 0
   menu .menubar.$x.m
   pack .menubar.$x -side left
   # Create the initial wild card entry for the component
   .menubar.$x.m add radio -label * \
      -variable font(cur,$x) \
      -value * \
      -command [list DoFont]
}
```

The menus for two components are left out. The two resolutions are virtually always the same, so one is enough. The avgWidth component varies wildly, and user probably won't choose a font based on it. Variable traces are used to fix up the values associated with these components. The second resolution is tied to the first resolution. The avgWidth always returns *, which matches anything. The points are set to 10 times the pixels if the pixels are set. However, if that isn't right, which sometimes happens, then the user can still set the points explicitly .

Example 24–7 Using variable traces to fix things up.

```
# Use traces to patch up the suppressed font(comps)
trace variable font(cur,res2) r TraceRes2
proc TraceRes2 { args } {
	global font
	set font(cur,res2) $font(cur,res)
}
trace variable font(cur,avgWidth) r TraceWidth
proc TraceWidth { args } {
	global font
	set font(cur,avgWidth) *
}
# Mostly, but not always, the points are 10x the pixels
trace variable font(cur,pixels) w TracePixels
proc TracePixels { args } {
	global font
	catch {
	 # Might not be a number
	 set font(cur,points) [expr 10*$font(cur,pixels)]
	}
}
```

The application displays a listbox with all the possible font names in it. If you click on a font name its font is displayed. The set of possible font names is obtained by running the *xlsfonts* program.

Example 24–8 Listing available fonts.

```
# Create a listbox to hold all the font names
frame .body
set font(list) [listbox .body.list \
	-setgrid true -selectmode browse \
	-yscrollcommand {.body.scroll set}]
scrollbar .body.scroll -command {.body.list yview}
pack .body.scroll -side right -fill y
pack .body.list -side left -fill both -expand true

# Clicking on an item displays the font
bind $font(list) <ButtonRelease-1> [list SelectFont
$font(list) %y]

# Use the xlsfonts program to generate a
# list of all fonts known to the server.
Status "Listing fonts..."
if [catch {open "|xlsfonts *"} in] {
	puts stderr "xlsfonts failed $in"
	exit 1
}
```

A simple data structure is created based on the list of available fonts. For each font component, all possible values are recorded. These values are used to create menus later on.

Example 24–9 Determining possible font components.

```
set font(num) 0
set numAliases 0
set font(N) 0
while {[gets $in line] >= 0} {
    $font(list) insert end $line
    # fonts(all,$i) is the master list of existing fonts
    # This is used to avoid potentially expensive
    # searches for fonts on the server, and to
    # highlight the matching font in the listbox
    # when a pattern is specified.
    set font(all,$font(N)) $line
    incr font(N)

    set parts [split $line -]
    if {[llength $parts] < 14} {
        # Aliases do not have the full information
        lappend aliases $line
        incr numAliases
    } else {
        incr font(num)
        # Chop up the font name and record the
        # unique font(comps) in the font array.
        # The leading - in font names means that
        # parts has a leading null element and we
        # start at element 1 (not zero).
        set i 1
        foreach x $font(comps) {
            set value [lindex $parts $i]
            incr i
            if {[lsearch $font($x) $value] < 0} {
                # Missing this entry, so add it
                lappend font($x) $value
            }
        }
    }
}
```

Menus are created so the user can select different font components. Radio button entries are used so that the current selection is highlighted. The special case for the two suppressed components crops up here. We let the variable traces fix up those values.

Example 24–10 Creating the `radiobutton` menu entries.

```
# Fill out the menus
foreach x $font(comps) {
```

```
    if {$x == "res2" || $x == "avgWidth"} {
     continue
    }
    foreach value [lsort $font($x)] {
       if {[string length $value] == 0} {
          set label (nil)
       } else {
          set label $value
       }
       .menubar.$x.m add radio -label $label \
          -variable font(cur,$x) \
          -value $value \
          -command DoFont
    }
}
Status "Found $font(num) fonts and $numAliases aliases"
```

Below the menu is a label that holds the current font name. Below that is a message widget that displays a sample of the font. One of two messages are displayed, depending on if the font is matched or not.

Example 24–11 Setting up the `label` and `message` widgets.

```
# This label displays the current font
label .font -textvar font(current) -bd 5 -font fixed

# A message displays a string in the font.
set font(msg) [message .font(msg) -aspect 1000 -borderwidth
10]
set font(sampler) "
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqurstuvwxyz
0123456789
!@#$%^&*()_+-=[]{};:'''~,.<>/?\\|
"
set font(errormsg) "

(No matching font)


"
# font Now pack the main display
pack .menubar -side top -fill x
pack .body -side top -fill both -expand true
pack .font $font(msg) -side top
```

The next example has the core procedures of the example. The `DoFont` procedure is triggered by changing a radio button menu entry . It rebuilds the font name and calls `SetFont`. The `SetFont` procedure searches the list of all fonts for a match. This prevents expensive searches by the X server , and it allows the application to highlight the matching font in the listbox. The `SelectFont` procedure is triggered by a selection in the listbox. It also constructs a font name and calls

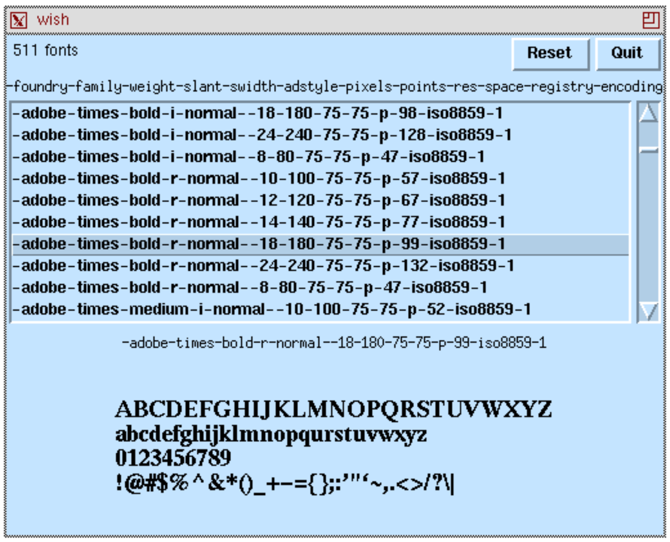
SetFont. Finally , Reset restores the font name to the match-all pattern.

Example 24–12 The font selection procedures.

```
proc DoFont { } {
    global font
    set font(current) {}
    foreach x $font(comps) {
     append font(current) -$font(cur,$x)
    }
    SetFont
}
proc SelectFont { list y } {
    # Extract a font name from the listbox
    global font
    set ix [$font(list) nearest $y]
    set font(current) [$font(list) get $ix]
    set parts [split $font(current) -]
    if {[llength $parts] < 14} {
        foreach x $font(comps) {
            set font(cur,$x) {}
        }
    } else {
        set i 1
        foreach x $font(comps) {
            set value [lindex $parts $i]
            incr i
            set font(cur,$x) $value
        }
    }
    SetFont
}
proc SetFont {} {
    global font
    # Generate a regular expression from the font pattern
    regsub -all -- {<nil>} $font(current) {} font(current)
    regsub -all -- {\*} $font(current) {[^-]*} pattern
    for {set n 0} {$n < $font(N)} {incr n} {
        if [regexp -- $pattern $font(all,$n)] {
            $font(msg) config -font $font(current) \
                -text $font(sampler)
            catch {$font(list) select clear \
                [$font(list) curselection]}
            $font(list) select set $n
            $font(list) see $n
            return
        }
    }
    $font(msg) config -text $font(errormsg)
}

proc Reset {} {
    global font
    foreach x $font(comps) {
```

```
        set font(cur,$x) *
    }
    DoFont
    Status "$font(num) fonts"
}

Reset
```

This is what the interface looks like.



CHAPTER 25

Window Managers and Window Information

A window manager is a special application that can control the size and location of other applications' windows. The `wm` command provides an interface to the window manager. The `winfo` command returns information about windows.

Management of toplevel windows is done by a distinguished application called the *window manager*. The window manager controls the position of toplevel windows, and it provides a way to resize windows, open and close them, and it implements a border and decorative title for windows. The window manager contributes to the general look and feel of the X display, but there is no requirement that the look and feel of the window manager be the same as that used in an application. The `wm` command is used to interact with the window manager so that the application itself can control its size, position, and iconified state.

If you need to fine tune your display you may need some detailed information about widgets. The `winfo` command returns all sorts of information about windows, including interior widgets, not just toplevel windows.

The wm Command

The wm command has about 20 operations that interact with the window manager. The general form of the commands is:

```
wm operation win ?args?
```

In all cases the *win* argument must be for a toplevel window. Otherwise an error is raised. In many cases the operation either sets or queries a value. If a new value is not specified, then the current settings are returned. For example, the first command below returns the current window geometry, and the next command defines a new geometry.

```
wm geometry .
=> 300x200+327+20
wm geometry . 400x200+0+0
```

The operations can be grouped into four main categories.

- Size, placement and decoration of windows.
- Icons.
- Long term session state.
- Miscellaneous.

Size, placement, and decoration

Perhaps the most commonly used `wm` operation is `wm title` that sets the title of the window. The title appears in the title bar that the window manager places above your application's main window. The title may also appear in the icon for your window, unless you specify another name with `wm iconname`.

```
wm title . "My Application"
```

The `wm geometry` command can be used to adjust the position or size of your main windows. A geometry specification has the general form *W*x*H*+*X*+*Y*, where *W* is the widget, *H* is the height, and *X* and *Y* specify the location of the upper-left corner of the window. The location `+0+0` is the upper-left corner of the display. You can specify a negative *X* or *Y* to position the bottom (right) side of the window relative to the bottom (right) side of the display. For example, `+0-0` is the lower left corner, and `-100-100` is offset from the lower-right corner by 100 pixels in the X and Y direction. If you do not specify a geometry, then the current geometry is returned.

A window can have a gridded geometry, which means that the geometry is in terms of some unit other than pixels. For example, the text and listbox widgets can set a grid based on the size of the characters they display. You can define a grid with the `wm grid` command, or you can use that command to find out what the current grid size is. The next example sets up gridded geometry for a canvas.

Example 25–1 Gridded geometry for a `canvas`.

```
canvas .c -width 300 -height 150
pack .c -fill both -expand true
wm geometry
=> 300x200+678+477
wm grid . 30 15 10 10
wm geometry .
=> 30x20+678+477
```

An important side effect of gridding is that it enables interactive resizing of windows. By default, Tk windows are not resizable except by program control.

You can constrain the minimum size, maximum size, and the aspect ratio of a toplevel window. The aspect ratio is the width divided by the height. The constraint is applied when the user resizes the window interactively. The `minsize`, `maxsize`, and `aspect` operations apply these constraints. As with gridding, a side effect of setting one of these constraints is to allow interactive resizing.

Some window managers insist on having the user position windows. The `sizefrom` and `positionfrom` operations let you pretend that the user specified the size and position in order to work around this restriction.

Table 25–1 summarizes the `wm` commands that deal with size, decorations, placement.

Table 25–1 Size, placement and decoration window manager operations.

`wm apsect` *`win`* `?`*`a b c d`*`?`	Constrain *`win`*'s ratio of width to height to be between (*`a/b`* and *`c/d`*).
`wm geometry` *`win`* `?`*`geometry`*`?`	Query or set the geometry of *`win`*.
`wm grid` *`win`* `?`*`w h dx dy`*`?`	Query or set the grid size. *`w`* and *`h`* are the base size, in grid units. *`dx`* and *`dy`* are the size, in pixels, of a grid unit.
`wm group` *`win`* `?`*`leader`*`?`	Query or set the group leader (a toplevel widget) for win. The window manager may unmap all the group at once.
`wm maxsize` *`win`* `?`*`width height`*`?`	Constrain the maximum size of *`win`*.
`wm minsize` *`win`* `?`*`width height`*`?`	Constrain the minimum size of *`win`*.
`wm positionfrom` *`win`* `?`*`who`*`?`	Query or set *`who`* to be `program` or `user`.
`wm sizefrom` *`win`* `?`*`who`*`?`	Query or set *`who`* to be `program` or `user`.
`wm title` *`win`* `?`*`string`*`?`	Query or set the window title to *`string`*.

Icons

When you close a window the window manager unmaps the window and replaces it with an icon. You can open and close the window yourself with the `deiconify` and `iconify` operations, respectively. Use the `withdraw` operation to unmap the window without replacing it with an icon. The `state` operation returns the current state, which is one of `normal`, `iconified`, or `withdrawn`. If you withdraw a window, you can restore it with `deiconify`.

You can set the attributes of the icon with the `iconname`, `iconposition`, `iconbitmap`, and `iconmask` operations. The icon's mask is used to get irregularly shaped icons. Chapter 23 describes how masks bitmaps are defined. In the case of an icon, it is most likely that you have the definition in a file, so your command will look like this:

```
wm iconbitmap . @myfilename
```

Table 25–2 summarizes the window manager commands that have to do with icons.

Table 25–2 Window manager commands for icons.

`wm deiconify` *win*	Open the window *win*.
`wm iconbitmap` *win* ?*bitmap*?	Query or define the bitmap for the icon.
`wm iconify` *win*	Close the window *win*.
`wm iconmask` *win* ?*mask*?	Query or define the mask for the icon.
`wm iconname` *win* ?*name*?	Query or set the name on the icon.
`wm iconposition` *win* ?*x y*?	Query or set the location of the icon.
`wm iconwindow` *win* ?*window*?	Query or specify an alternate window to display when in the iconified state.
`wm state` *win*	Returns `normal`, `iconic`, or `withdrawn`.
`wm withdraw` *win*	Unmap the window and forget about it. No icon is displayed.

Session state

Some window managers support the notion of a *session* that lasts between runs of the window system. A session is implemented by saving state about the applications that are running, and using this information to restart the applications when the window system is restarted. This section also describes how you can intercept requests to quit your application so you can stop cleanly.

An easy way to participate in the session protocol is to save the command used to start up your application. The `wm command` operation does this. The *wish* shell saves this information, so it is just a matter of registering it with the window manager. `argv0` is the command, and `argv` is the command line arguments.

```
wm command . "$argv0 $argv"
```

If your application is typically run on a different host than the one with the display (like in an Xterminal environment), then you also need to record what host to run the application on. Use the `wm client` operation for this. You may need to use `uname -n` instead of `hostname` on your system.

```
wm client . [exec hostname]
```

The window manager usually provides a way to quit applications. If you have any special processing that needs to take place when the user quits, then you need to intercept the quit action. Use the `wm protocol` operation to register a command that handles the `WM_DELETE_WINDOW` message from the window manager. The command must eventually call `exit` to actually stop your application.

```
wm protocol . WM_DELETE_WINDOW Quit
```

Other window manager messages that you can intercept are `WM_SAVE_YOURSELF` and `WM_TAKE_FOCUS`. The first is called by some session managers when shutting down. The latter is used in the active focus model. Tk (and this book) assumes a passive focus model where the window manager assigns focus to a toplevel window .

describes the session-related window manager operations.

Table 25–3 Session-related window manager operations.

`wm client` *`win`* `?`*`name`*`?`	Record the hostname in the `WM_CLIENT_MACHINE` property .
`wm command` *`win`* `?`*`command`*`?`	Record the startup command in the `WM_COMMAND` property .
`wm protocol` *`win`* `?`*`name`*`? ?`*`command`*`?`	Register a *`command`* to handle the protocol request *`name`*, which can be `WM_DELETE_WINDOW`, `WM_SAVE_YOURSELF`, `WM_TAKE_FOCUS`.

Miscellaneous

A window manager works by reparenting an applications window so it is a child of the window that forms the border and decorative title bar . The `wm frame` operation returns the window ID of the new parent, or the id of the window itself if it has not been reparented. The `winfo id` command returns the id of a window . The `wm overrideredirect` operation can set a bit that overrides the reparenting. This means that no title or border will be drawn around the window , and you cannot control the window through the window manager .

The `wm group` operation is used to collect groups of windows so that the window manager can open and close them together . Not all window managers implement this. One window , typically the main window , is chosen as the leader . The other members of the group are iconified when it is iconified.

The wm transient operation informs the window manager that this is a temporary window and there is no need to decorate it with the border and decorative title bar . This is used, for example, on pop-up menus, but in that case it is handled by the menu implementation.

Table 25–4 lists the remaining window manager operations.

Table 25–4 Miscellaneous window manager operations.

`wm focusmodel` *`win`* `?`*`what`*`?`	Set the focus model to `active` or `passive`. Many parts of Tk assume the `passive` model.
`wm frame` *`win`*	Return the ID of the parent of *`win`* has been reparented, otherwise return the ID of *`win`* itself.
`wm group` *`win`* `?`*`leader`*`?`	Assign *`win`* to the group headed by *`leader`*.
`wm overrideredirect` *`win`* `?`*`boolean`*`?`	Set the override redirect bit that suppresses reparenting by the window manager .
`wm transient` *`win`* `?`*`leader`*`?`	Query or mark a window as transient window working for *`leader`*, another widget.

The winfo Command

The winfo command has just over 40 operations that return information about a widget or the display . The operations fall into the following categories.

- Sending commands between applications.
- Family relationships.
- Size.
- Location.
- Virtual root coordinates.
- Atoms and IDs.
- Colormaps and visuals.

Sending commands between applications

Each Tk application has a name that is used when sending commands between applications using the `send` command. The list of Tk applications is returned by the `interps` operation. The `tk appname` is used to get the name of the application, and that command can also be used to set the application name. In Tk 3.6 and earlier , you had to use `winfo name .` to get the name of the application.

Example 25–2 Telling other applications what your name is.

```
foreach app [winfo interps] {
   catch {send $app [list Iam [tk appname .]]}
}
```

The example shows how your application might connect up with several existing applications. It contacts each registered Tk interpreter and sends a short command that contains the applications own name as a parameter . The other application can use that name to communicate back.

Table 25–5 summarizes these commands.

Table 25–5 Information useful with the `send` command.

tk appname ? *newname*?	Query or set the name used with `send`.
winfo name .	Also returns the name used for `send`, for backward compatibility with Tk 3.6 and earlier .
winfo name *pathname*	Return the last component of *pathname*.
winfo interps	Return the list of registered Tk applications.

Family relationships

The Tk widgets are arranged in a hierarchy , and you can use the winfo command to find out about the structure of the hierarchy. The `winfo children` operation returns the children of a window, and the `winfo parent` operation returns the parent. The parent of the main window is null (i.e., an empty string).

A widget is also a member of a class, which is used for bindings and as a key into the X resource database. The `winfo class` operation returns this information. You can test for the existence of a window with `window exists`, and whether or not a window is mapped onto the screen with `winfo ismapped`.

The `winfo manager` operation tells you what geometry manager is controlling the placement of the window. This returns the name geometry manager command. Examples include `pack`, `place`, `canvas`, and `text`. The last two indicate the widget is imbedded into a canvas or text widget.

Table 25–5 summarizes these winfo operations.

Table 25–6 Information about the window hierarchy.

`winfo children` *`win`*	Return the list of children widgets of *`win`*.
`winfo class` *`win`*	Return the binding and resource class of *`win`*.
`winfo exists` *`win`*	Returns 1 if the *`win`* exists.
`winfo ismapped` *`win`*	Returns 1 if *`win`* is mapped onto the screen.
`winfo manager` *`win`*	The geometry manager: `pack place canvas text`
`winfo parent` *`win`*	Returns the parent widget of *`win`*.

Size

The `winfo width` and `winfo height` operations return the width and height of a window, respectively. However, a window's size is not set until a geometry manager maps a window onto the display. Initially a window starts out with a width and height of 1. You can use `tkwait visibility` to wait for a window to be mapped before asking its width or height.

Alternatively, you can ask for the requested width and height of a window. Use `winfo reqwidth` and `winfo reqheight` for this information. The requested size may not be accurate, however, because the geometry manager may allocate more of less space, and the user may resize the window.

The `winfo geometry` operation returns the size and position of the window in the standard geometry format: *`W`*`x`*`H`*`+`*`X`*`+`*`Y`*. In this case the X and Y offsets are relative to the parent widget, or relative to the root window in the case of the main window.

You can find out how big the display is, too. The `winfo screenwidth` and winfo `screenheight operations` return this information in pixels. The `winfo screenmmwidth` and `winfo screenmmheight` return this information in millimeters.

You can convert between pixels and screen distances with the `winfo pixels` and `winfo fpixels` operations. Given a number of screen units such as `10m`, `3c`, or `72p`, these return the corresponding number of pixels. The first form rounds to a whole number , while the second form returns a floating point number . Chapter 22 has an explanation of the screen units. For example:

```
set pixelsToInch [winfo pixels . 2.54c]
```

Table 25–5 summarizes these operations.

Table 25–7 Information about the window size.

`winfo fpixels` *`win num`*	Convert *num*, in screen units, to pixels. Returns a floating point number .
`winfo geometry` *`win`*	Return the geometry of *win*, in pixels and relative to the parent in the form *W*x*H*+*X*+*Y*
`winfo height` *`win`*	Return the height of *win*, in pixels.
`winfo pixels` *`win num`*	Convert num to a whole number of pixels.
`winfo reqheight` *`win`*	Return the requested height of *win*, in pixels.
`winfo reqwidth` *`win`*	Return the requested width of *win*, in pixels.
`winfo screenheight` *`win`*	Return the height of the screen, in pixels.
`winfo screenmmheight` *`win`*	Return the height of the screen, in millimeters.
`winfo screenmmwidth` *`win`*	Return the width of the screen, in millimeters.
`winfo screenwidth` *`win`*	Return the width of the screen, in pixels.
`winfo width` *`win`*	Return the width of *win*, in pixels.

Location

The `winfo x` and `winfo y` operations return the position of a window relative to its parent widget. In the case of the main window , this is its location on the screen. The `winfo rootx` and `winfo rooty` return the location of a widget on the screen, even if it is not a toplevel window .

The `winfo containing` operation returns the pathname of the window that contains a point on the screen. This is useful in implementing menus and drag and drop applications.

The `winfo toplevel` operation returns the pathname of the toplevel window that contains a widget. If the window is itself a toplevel, then this operation returns its pathname.

The `winfo screen` operation returns the display identifier for the screen of the window . This value is useful in the `selection` command.

Table 25–5 summarizes these operations.

Table 25–8 Information about the window location.

`winfo containing` *`win x y`*	Return the pathname of the window at *x* and *y*.
`winfo rootx` *`win`*	Return the X screen position of *win*.

Table 25–8 Information about the window location.

`winfo rooty` *`win`*	Return the Y screen position of *`win`*.
`winfo screen` *`win`*	Return the display identifier of *`win`*'s screen.
`winfo toplevel` *`win`*	Return the pathname of the toplevel that contains *`win`*.
`winfo x` *`win`*	Return the X position of *`win`* in its parent.
`winfo y` *`win`*	Return the Y position of *`win`* in its parent.

Virtual root window

A virtual root window is used by some window managers to give the user a larger virtual screen. At any given time only a portion of the virtual screen is visible, and the user can change the view on the virtual screen to bring different applications into view. In this case, the `winfo x` and `winfo y` operations return the coordinates of a main window in the virtual root window (i.e., not the screen).

The `winfo vrootheight` and `winfo vrootwidth` operations return the size of the virtual root window. If there is no virtual root window, then these just return the size of the screen.

The `winfo vrootx` and `winfo vrooty` are used to map from the coordinates in the virtual root window to screen-relative coordinates. These operations return 0 if there is no virtual root window. Otherwise they return a negative number. If you add this number to the value returned by `winfo x` or `winfo y`, it gives the screen-relative coordinate of the window.

Table 25–5 summarizes these operations.

Table 25–9 Information associated with virtual root windows.

`winfo containing` *`win x y`*	Return the pathname of the window at *x* and *y*.
`winfo rootx` *`win`*	Return the X screen position of *`win`*.
`winfo rooty` *`win`*	Return the Y screen position of *`win`*.
`winfo screen` *`win`*	Return the display identifier of *`win`*'s screen.
`winfo toplevel` *`win`*	Return the pathname of the toplevel that contains *`win`*.
`winfo x` *`win`*	Return the X position of *`win`* in its parent.
`winfo y` *`win`*	Return the Y position of *`win`* in its parent.

Atoms and IDs

An *atom* is an X technical term for an identifier that is registered with the X server. Applications map names into atoms, and the X server assigns each atom a 32 bit identifier that can then be passed around. One of the few places this is used in Tk is when the selection mechanism is used to interface with different toolkits. In some cases the selection is returned as atoms, which appear as 32 bit integers. The `winfo atomname` operation converts that number into an atom (i.e., a string), and the `winfo atom` registers a string with the X server and returns the 32-bit identifier as a hexadecimal string.

Each widget has an ID from the X server . The `winfo id` command returns this identifier . The `winfo pathname` operation returns the Tk pathname of the widget that has a given ID, but only if the window is part of the same application.

Table 25–5 summarizes these operations.

Table 25–10 Information about atoms and window ids.

`winfo atom` *`name`*	Returns the 32-bit identifier for the atom *`name`*.
`winfo atomname` *`id`*	Returns the atom that corresponds to the 32-bit ID.
`winfo id` *`win`*	Returns the X window ID of *`win`*.
`winfo pathname` *`id`*	Returns the Tk pathname of the window with *`id`*, or null.

Colormaps and visuals

Chapter 23 describes colormaps and visual classes in detail. The `winfo depth` returns the number of bits used to represent the color in each pixel. The `winfo cells` command returns the number of colormap entries used by the visual class of a window . These two values of generally related. A window with 8 bits per pixel usually has 256 colormap cells. The `winfo screendepth` and `winfo screencells` return this information for the default visual class of the screen.

The `winfo visualsavailable` command returns a list of the visual classes and screen depths that are available. For example, a display with 8 bits per pixel might report the following visual classes are available:

```
winfo visualsavailable .
=> {staticgray 8} {grayscale 8} {staticcolor 8} \
     {pseudocolor 8}
```

The `winfo visual` operation returns the visual class of a window , and the `winfo screenvisual` returns the default visual class of the screen.

The winfo rgb operation converts from a color name or value to the red, green, and blue components of that color . Three decimal values are returns. Example 23–2 uses this command to compute a slightly darker version of the same color .

Table 25–5 summarizes these operations.

Table 25–11 Information about colormaps and visual classes.

`winfo cells` *`win`*	Returns the number of colormap cells in *`win`*'s visual.
`winfo depth` *`win`*	Return the number of bits per pixel for *`win`*.

Table 25–11 Information about colormaps and visual classes.

`winfo rgb` *`win color`*	Return the red, green, and blue values for *`color`*.
`winfo screencells` *`win`*	Returns the number of colormap cells in the default visual.
`winfo screendepth` *`win`*	Returns the number of bits per pixel in the screen's default visual.
`winfo visual` *`win`*	Returns the visual class of win.
`winfo visualsavailable` *`win`*	Returns a list of pairs that specify the visual type and bits per pixel of the available visual classes.

The tk Command

The `tk` command provides a few miscellaneous entry points into the Tk library. The first form is used to set or query the application name used with the Tk send command. If you define a new name and it is already in use by another application, (perhaps another instance of yourself) then a number is appened to the name (e.g., `#2`, `#3`, and so on).

```
tk appname ?name?
```

The other form of the `tk` command is used to query and set the *colormodel* of the application. The colormodel is either `monochrome` or `color`, and it determines what default colors are chosen the the Tk widgets. You should test the colormodel yourself before setting up colors in your application. Note that when a color allocation fails, Tk automatically changes the colormodel to `monochrome`. You can force it back into color mode with another call to `tk colormodel`. This form of the command is shown below.

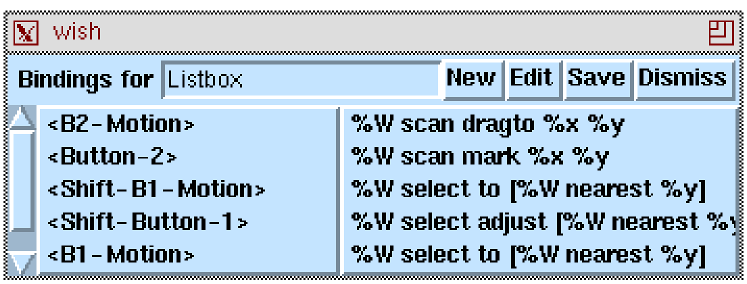
```
tk colormodel window ?what?
```

C H A P T E R 26

A User Interface to bind

This chapter presents a user interface to view and edit bindings.

A good way to learn about how a widget works is to examine the bindings that are defined for it. This chapter presents a user interface that lets you browse and change bindings for a widget or a class of widgets. Here is what the display looks like.



A Binding User Interface

The interface uses a pair of listboxes to display the events and their associated commands. An entry widget is used to enter the name of a widget or a class. There are a few command buttons that let the user add a new binding, edit an existing binding, save the bindings to a file, and dismiss the dialog.

Example 26–1 A user interface to widget bindings.

```
proc Bind_Interface { w } {
	# Our state
	global bind
	set bind(class) $w

	# Set a class used for resource specifications
	set frame [toplevel .bindui -class Bindui]
	# Default relief
	option add *Bindui*Entry.relief sunken startup
	option add *Bindui*Listbox.relief raised startup
	# Default Listbox sizes
	option add *Bindui*key.width 18 startup
	option add *Bindui*cmd.width 25 startup
	option add *Bindui*Listbox.height 5 startup

	# A labeled entry at the top to hold the current
	# widget name or class.
	set t [frame $frame.top -bd 2]
	label $t.l -text "Bindings for"
	entry $t.e -textvariable bind(class)
	pack $t.l -side left
	pack $t.e -side left -fill x -expand true
	pack $t -side top -fill x

	bind $t.e <Return> [list Bind_Display $frame]

	# Command buttons
	button $t.quit -text Dismiss \
		-command [list destroy $frame]
	button $t.save -text Save \
		-command [list Bind_Save $frame]
	button $t.edit -text Edit \
		-command [list Bind_Edit $frame]
	button $t.new -text New \
		-command [list Bind_New $frame]
	pack $t.quit $t.save $t.edit $t.new -side right

	# A pair of listboxes and a scrollbar
	scrollbar $frame.s -orient vertical \
		-command [list BindYview \
			[list $frame.key $frame.cmd]]
	listbox $frame.key \
		-yscrollcommand [list $frame.s set] \
		-exportselection false
	listbox $frame.cmd \
		-yscrollcommand [list $frame.s set]
	pack $frame.s -side left -fill y
	pack $frame.key $frame.cmd -side left \
		-fill both -expand true
```

```
    foreach l [list $frame.key $frame.cmd] {
        bind $l <B2-Motion>\
            [list BindDragto %x %y $frame.key $frame.cmd]
        bind $l <Button-2> \
            [list BindMark %x %y $frame.key $frame.cmd]
        bind $l <Button-1> \
            [list BindSelect %y $frame.key $frame.cmd]
        bind $l <B1-Motion> \
            [list BindSelect %y $frame.key $frame.cmd]
        bind $l <Shift-B1-Motion> {}
        bind $l <Shift-Button-1> {}
    }
    # Initialize the display
    Bind_Display $frame
}
```

The `Bind_Interface` command takes a widget name or class as a parameter. It creates a toplevel window and gives it the Bindui class so that X resources can be set to control widget attributes. The `option add` command is used to set up the default listbox sizes. The lowest priority, `startup`, is given to these resources so that clients of the package can override the size with their own resource specifications.

At the top of the interface is a labeled entry widget. The entry holds the name of the class or widget for which the bindings are displayed. The `textvariable` option of the entry widget is used so that the entry's contents are available in a variable, `bind(class)`. Pressing `<Return>` in the entry invokes `Bind_Display` that fills in the display.

Example 26–2 Bind_Display presents the bindings for a given widget or class.

```
proc Bind_Display { frame } {
    global bind
    $frame.key delete 0 end
    $frame.cmd delete 0 end
    foreach seq [bind $bind(class)] {
        $frame.key insert end $seq
        $frame.cmd insert end [bind $bind(class) $seq]
    }
}
```

The `Bind_Display` procedure fills in the display with the binding information. It used the `bind` command to find out what events have bindings, and what the command associated with each event is. It loops through this information and fills in the listboxes.

A Pair of Listboxes Working Together

The two listboxes in the interface, `$frame.key` and `$frame.cmd`, are set up to work as a unit. A selection in one causes a parallel selection in the other. A single scrollbar scrolls both of them. This is achieved with some simple bindings that accept a variable number of arguments. The first arguments are coordinates, and then the rest are some number of listboxes that need to be operated on as a group.

Example 26–3 Related listboxes are configured to select items together.

```
foreach l [list $frame.key $frame.cmd] {
   bind $l <Button-1> \
      [list BindSelect %y $frame.key $frame.cmd]
   bind $l <B1-Motion> \
      [list BindSelect %y $frame.key $frame.cmd]
}
proc BindSelect { y args } {
   foreach w $args {
      $w select clear 0 end
      $w select anchor [$w nearest $y]
      $w select set anchor [$w nearest $y]
   }
}
```

The `bind` commands from `Bind_Interface` are repeated in the example. The `BindSelect` routine selects an item in both listboxes. In order to have both selections highlighted, the listboxes are prevented from exporting their selection as the X `PRIMARY` selection. Otherwise, the last listbox to assert the selection would steal the selection rights away from the first widget.

A single scrollbar is created and set up to control both listboxes.

Example 26–4 Controlling a pair of listboxes with one scrollbar.

```
scrollbar $frame.s -orient vertical \
   -command [list BindYview [list $frame.key $frame.cmd]]

proc BindYview { lists args } {
   foreach l $lists {
      eval {$l yview} $args
   }
}
```

The `scrollbar` command from the `Bind_Interface` procedure is repeated in the example. The `BindYview` command is used to change the display of the listboxes associated with the scrollbar. Before the scroll command is evaluated some additional parameters are added that specify how to position the display. The details are essentially private between the scrollbar and the listbox, so the `args` keyword is used to represent these extra arguments, and `eval` is used to pass them through `BindYview`. The reasoning for using `eval` like this is explained in Chapter 6 in the section on *Eval And Concat*.

The `Listbox` class bindings for `<Button-2>` and `<B2-Motion>` cause the listbox to scroll as the user drags the widget with the middle mouse button. These bindings are adjusted in the example so that both listboxes move together.

Example 26–5 Drag-scrolling a pair of listboxes together.

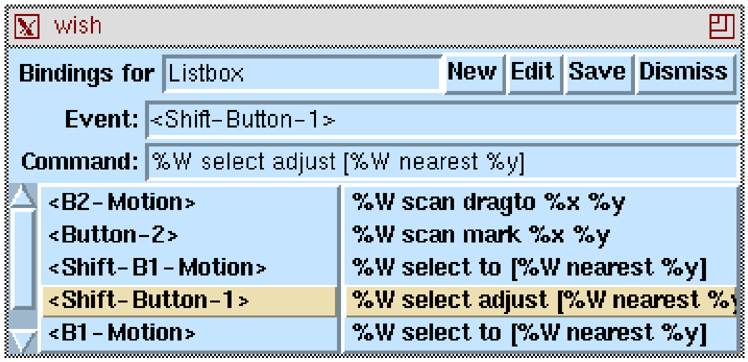
```
bind $l <B2-Motion>\
    [list BindDragto %x %y $frame.key $frame.cmd]
bind $l <Button-2> \
    [list BindMark %x %y $frame.key $frame.cmd]

proc BindDragto { x y args } {
    foreach w $args {
        $w scan dragto $x $y
    }
}
proc BindMark { x y args } {
    foreach w $args {
        $w scan mark $x $y
    }
}
```

The `bind` commands from the `Bind_Interface` procedure are repeated in this example. The `BindMark` procedure does a `scan mark` that defines an origin, and `BindDragto` does a `scan dragto` that scrolls the widget based on the distance from that origin. All Tk widgets that scroll support `yview`, `scan mark`, and `scan dragto`. Thus the `BindYview`, `BindMark`, and `BindDragto` procedures are general enough to be used with any set of widgets that scroll together.

The Editing Interface

Editing and defining a new binding is done in a pair of entry widgets. These widgets are created and packed into the display dynamically when the user presses the `New` or `Edit` button.

Example 26–6 An interface to define bindings.




```
proc Bind_New { frame } {
    if [catch {frame $frame.edit} f] {
        # Frame already created
        set f $frame.edit
    } else {
        foreach x {key cmd} {
            set f2 [frame $f.$x]
            pack $f2 -fill x
            label $f2.l -width 9 -anchor e
            pack $f2.l -side left
            entry $f2.e
            pack $f2.e -side left -fill x -expand true
            bind $f2.e <Return> [list BindDefine $f]
        }
        $f.key.l config -text Event:
        $f.cmd.l config -text Command
    }
    pack $frame.edit -after $frame.top -fill x
}
proc Bind_Edit { frame } {
    Bind_New $frame
    set line [$frame.key curselection]
    if {$line == {}} {
        return
    }
    $frame.key.e delete 0 end
    $frame.key.e insert 0 [$frame.key get $line]
    $frame.cmd.e delete 0 end
    $frame.cmd.e insert 0 [$frame.cmd get $line]
}
```

The `-width 9` and `-anchor e` attributes for the label widgets are specified so that the `Event:` and `Command:` labels will line up properly. Another approach would be to fix the width of the entry widgets, and then use the `-anchor` option when packing the labels.

```
label $f2.l
entry $f2.e -width 44
pack $f2.e -side right
pack $f2.l -side right -anchor e
```

All that remains is the actual change or definition of a binding, and some way to remember the bindings the next time the application is run. A simple technique is to write out the definitions as a series of Tcl commands that define them.

Example 26–7 Defining and saving bindings.

```
proc BindDefine { f } {
    if [catch {
        bind [$f.top.e get] [$f.edit.key.e get] \
            [$f.edit.cmd.e get]
    } err] {
        Status $err
    } else {
        # Remove the edit window
        pack forget $f.edit
    }
}
proc Bind_Save { dotfile args } {
    set out [open $dotfile.new w]
    foreach w $args {
        foreach seq [bind $w] {
            # Output a Tcl command
            puts $out [list bind $w $seq [bind $w $seq]]
        }
    }
    close $out
    exec mv $dotfile.new $dotfile
}
proc Bind_Read { dotfile } {
    if [catch {
        if [file exists $dotfile] {
            # Read the saved Tcl commands
            source $dotfile
        }
    } err] {
        Status "Bind_Read $dotfile failed: $err"
    }
}
```

The `BindDefine` procedure attempts a bind command that uses the contents of the entries. If it succeeds, then the edit window is removed by unpacking it. The `Bind_Save` procedure writes a series of Tcl commands to a file. It is crucial that the `list` command be used to construct the command properly. Finally, `Bind_Read` uses the source command to read the saved commands.

The application will have to call `Bind_Read` as part of its initialization in order to get the customized bindings for the widget or class. It will also have to provide a way to invoke `Bind_Interface`, such as a button, menu entry, or key binding.

CHAPTER 27

Using X Resources

This chapter describes the use of the X resource database. It describes a way for users to define buttons and menu via resource specifications.

X supports a resource database through which your application can be customized by users and site administrators. The database holds specifications of widget attributes such as fonts and colors. You can control all attributes of the Tk widgets through the resource database. It can also be used as a more general database of application-specific parameter settings.

Because a Tk application can use Tcl for customization, it might not seem necessary to use the X resource mechanism. However, partly because users have grown to expect it, and partly because of the flexibility it provides, the X resource mechanism is a useful tool for your Tk application.

An Introduction To X Resources

When a Tk widget is created, its attributes are set by one of three sources. the most evident source is the command line switches in the tcl command, such as the `-text quit` attribute specification for a button. If an attribute is not specified on the command line, then the X resource database is queried as described below. Finally, if there is nothing in the resource database, then a hardcoded value from the widget implementation is used. It is important to note that command line specifications have priority over resource database specifications.

The resource database consists of a set of keys and values. Unlike many databases, however, the keys are patterns that are matched against the names of widgets and attributes. This makes it possible to specify attribute values for a large number of widgets with just a few database entries. In addition, the resource database can be shared by many applications, so users and administrators can define common attributes for their whole set of applications.

The resource database is maintained in main memory by the Tk toolkit. It is initialized from your `~/.xdefaults` file, and from additional files that are explicitly loaded by the Tk application.* You can also add individual database entries with the `option` tcl command.

The pattern language for the keys is related to the naming convention for tk widgets. Recall that a widget name reflects its position in the hierarchy of windows. You can think of the resource names as extending the hierarchy one more level at the bottom to account for all the attributes of each individual widget. There is also a new level of the hierarchy at the top in order to specify the application by name. For example, the database could contain an entry like the following in order to define a font for the quit button in a frame called `.buttons`.

```
Tk.buttons.quit.font: fixed
```

The leading `Tk.` matches the default class name for `wish` applications. You could also specify a more specific application name, such as `exmh`, or an asterisk to match any application.

Resource keys can also specify *classes* of widgets and attributes as opposed to individual instances. The quit button, for example, is an instance of the Button class. Class names for widgets are the same as the tcl command used to create them, except for a leading capital. A class-oriented specification that would set the font for all buttons in the `.buttons` frame would be:

```
Tk.buttons.Button.font: fixed
```

Patterns allow you to replace one or more components of the resource name with an asterisk (*). For example, to set the font for all the widgets packed into the `.buttons` frame, you could use the resource name `*buttons*font`. Or, you could specify the font for all buttons with the pattern `*Button.font`. In these examples we have replaced the leading Tk with an asterisk as well. It is the ability to collapse several layers of the hierarchical name with a single asterisk that makes it easy to specify attributes for many widgets with just a few database entries.

You can determine the resource names for the attributes of different widgets by consulting their man page, or by remembering the following convention. The resource name is the same as the command line switch (without the leading dash), except that multi-word attributes use a capital letter at the internal word boundaries. For example, if the command line switch is `-offvalue`, then the corresponding resource name is `offValue`. There are also class names for attributes, which are also distinguished with a leading capital (e.g., `OffValue`).

* This is a bit different than the Xt toolkit that loads specifications from as many as 5 different files to allow for per-user, per-site, per-application, per-machine, and per-user-per-application specifications.

Warning: order is important!

The matching between a widget name and the patterns in the database can be ambiguous. It is possible that multiple patterns can match the same widget. The way this is resolved in Tk is by the ordering of database entries, with later entries taking precedence.* Suppose the database contained just two entries, in this order.

```
*Text*foreground: blue
*foreground: red
```

In spite of the more specific `*Text*foreground` entry, all widgets will have a red foreground, even `text` widgets. For this reason you should list your most general patterns early in your resource files, and give the more specific patterns later.

Loading Option Database Files

The `option` command is used to manipulate the resource database. The first form of the command is used to load a file containing database entries.

```
option readfile filename ?priority?
```

The *priority* can be used to distinguish different sources of resource information and give them different priorities. From lowest to highest, the priorities are: `widgetDefault`, `startupFile`, `userdDefault`, `interactive`. These names can be abbreviated. The default priority is `interactive`.

Example 27–1 Reading an option database file.

```
if [file exists $appdefaults] {
    if [catch {option readfile $appdefaults startup} err] {
        puts stderr "error in $appdefaults: $err"
    }
}
```

The format of the entries in the file is:

```
key: value
```

The key has the pattern format described above. The value can be anything, and there is no need to group multi-word values with any quoting characters. In fact, quotes will be picked up as part of the value.

Comment lines are introduced by the exclamation character (!).

Example 27–2 A file containing resource specifications.

```
!
! Grey color set
```

* This is unlike other toolkits that use the length of a pattern where longer matching patterns have precedence, and instance specifications have priority over class specifications. (This may change in Tk 4.0).

```
! Slightly modified from Ron Frederick's nv grey family
!
*activeBackground: white
*activeForeground: black
*selectColor: black
*background: #efefef
*foreground: black
*selectBackground: #bfdfff
*troughColor: #efefef
*Scrollbar.background: #dfdfdf
*Scale.background: #dfdfdf
*disabledforeground: #7f7f7f
```

The example resource file specifies an alternate color scheme for the Tk widget set that is based on a family of gray levels. Color highlighting shows up well against this backdrop. Most of these colors are applied generically to all the widgets (e.g., `*background`), while there are a few special cases for `scale` and `scrollbar` widgets. The hex values for the colors specify 2 digits (8 bits) each for red, green, and blue.

Adding Individual Database Entries

You can enter individual database entries with the `option add` Tcl command. This is appropriate to handle special cases, or if you do not want to manage a separate per-application resource specification file. The command syntax is:

```
option add pattern value ?priority?
```

The *priority* is the same as that used with `option readfile`. The *pattern* and *value* are the same as in the file entries, except that the key does not have a trailing colon when specified in an `option add` command. Some of the specifications from the last example could be added as follows:

```
option add *foreground black
option add *Scrollbar.background #dfdfdf
```

You can clear out the option database altogether with:

```
option clear
```

Accessing The Database

Often it is sufficient to just set up the database and let the widget implementations use the values. However, it is also possible to record application-specific information in the database. To fetch a resource value, use `option get`:

```
option get window name class
```

The *window* is a Tk widget pathname. The *name* is a resource name. In this case, it is not a pattern or a full name. Instead, it is the simple resource name as specified in the man page. Similarly, the *class* is a simple class name. It is possible to specify a null name or class. If there is no matching database entry, `option` `get` returns the empty string.

User Defined Buttons

In a big application there might be many functions provided by various menus, but suppose we wanted users to be able to define a set of their own buttons for frequently executed commands. Or, as we will describe later, perhaps users can augment the application with their own Tcl code. The following scheme lets them define buttons to invoke their own code or their favorite commands.*

The user interface will create a special frame to hold the user-defined buttons, and place it appropriately. Assume the frame is created like this:

```
frame .user -class User
```

The class specification for the frame means that we can name resources for the widgets inside the frame relative to `*User`. Users will specify the buttons that go in the frame via a personal file containing X resource specifications.

The first problem is that there is no means to enumerate the database, so we must create a resource that lists the names of the user defined buttons. We will use the name `buttonlist`, and make an entry for `*user.buttonlist` that specifies what buttons are being defined. It is possible to use artificial resource names like this, but they must be relative to an existing Tk widget.

Example 27–3 Using resources to specify user-defined buttons.

```
*User.buttonlist: save search justify quit
*User.save.text: Save
*User.save.command: File_Save
*User.search.text: Search
*User.search.command: Edit_Search
*User.justify.text: Justify
*User.justify.command: Edit_Justify
*user.quit.text: Quit
*User.quit.command: File_Quit
*User.quit.background: red
```

In this example we have listed four buttons and specified some of the attributes for each, most importantly the `text` and `command` attributes. We are assuming, of course, that the application manual publishes a set of commands that users can invoke safely. In this simple example the commands are all one word, but there is no problem with multi-word commands. There is no interpretation done of the value, so it can include references to Tcl variables and nested command calls. The code that uses these resource specifications to define the buttons is given below.

* Special thanks go to John Robert LoVerso for this idea.

Example 27–4 Defining buttons from the resource database.

```
proc ButtonResources { f class } {
	frame $f -class $class -borderwidth 2
	pack $f -side top
	foreach b [option get $f buttonlist {}] {
		if [catch {button $f.$b}] {
			button $f.$b -font fixed
		}
		pack $f.$b -side right
	}
}
```

The `catch` phrase is introduced to handle a common problem with fonts and widget creation. If the user's resources specify a bogus or missing font, then the widget creation command will fail. The `catch` phrase guards against this case by falling back to the `fixed` font, which is guaranteed by the X server to always exist.

The button specification given in the previous example results in the display shown below.


```
option readfile button.resources
ButtonResources .user user
```

User Defined Menus

User-defined menus can be set up with a similar scheme. However, it is a little more complex because there are no resources for specific menu entries. We have to use some more artificial resources to emulate this. First use `menulist` to name the set of menus. Then for each of these we define an `entrylist` resource. Finally, for each entry we define a few more resources for the label, command, and menu entry type. The conventions will be illustrated by the next example.

Example 27–5 Specifying menu entries via resources.

```
*User.menulist: stuff
*User.stuff.text: My stuff
*User.stuff.m.entrylist: keep insert find
*User.stuff.m.l_keep: Keep on send
*User.stuff.m.t_keep: check
*User.stuff.m.v_keep: checkvar
*User.stuff.m.l_insert: Insert File...
*User.stuff.m.c_insert: InsertFileDialog
```

```
*User.stuff.m.l_find: Find
*User.stuff.m.t_find: cascade
*User.stuff.m.m_find: find
*User.stuff.m.find.entrylist: next prev
*User.stuff.m.find.l_next: Next
*User.stuff.m.find.c_next: Find_Next
*User.stuff.m.find.l_prev: Previous
*User.stuff.m.find.c_prev: Find_Previous
```

The menu structure created from the resource specification is shown below .

In the example, stuff is defined to be a menu. Actually , .user.stuff is a Tk menubutton. It has a menu as its child, .user.stuff.m, where the .m is set up by convention. You will see this in the code for MenuResources below. The entrylist for the menu is similar in spirit to the buttonlist resource. For each entry , however , we have to be a little creative with the next level of resource names. The following resources are used to specify the first entry in the menu:

```
*User.stuff.m.l_keep: Keep on send
*User.stuff.m.t_keep: check
*User.stuff.m.v_keep: checkvar
```

The l_*entryname* resource specifies the label (text) for the entry . The t_*entryname* resource specifies the type of the entry , which is a command entry by default. In this case we are defining a checkbutton entry . Associated with a checkbutton entry is a variable, which is defined with the v_*entryname* resource.

The insert menu entry is simpler , just requiring a label resource, l_insert, and a command resource, c_insert:

```
*user.stuff.m.l_insert: insert file...
*user.stuff.m.c_insert: insertfiledialog
```

The find menu entry is for a cascaded menu. This requires the type, t_find, to be cascade. Associated with a cascade entry is a submenu, m_find, and again a label, l_find:

```
*User.stuff.m.l_find: find
*User.stuff.m.t_find: cascade
*User.stuff.m.m_find: find
```

The conventions are the same for the cascaded menus, with *user.stuff.m.find.entrylist defining the entries, and so on. The code to support all this is given in the next example.

Example 27–6 Defining menus from resource specifications.

```
proc MenuResources { f class } {
    set f [frame .user -class User]
    pack $f -side top
    foreach b [option get $f menulist {}] {
        set cmd [list menubutton $f.$b -menu $f.$b.m \
                    -relief raised]
        if [catch $cmd t] {
            eval $cmd {-font fixed}
        }
        if [catch {menu $f.$b.m}] {
            menu $f.$b.m -font fixed

        pack $f.$b -side right
        MenuButtonInner $f.$b.m
    }
}
proc MenuButtonInner { menu } {
    foreach e [option get $menu entrylist {}] {
        set l [option get $menu l_$e {}]
        set c [option get $menu c_$e {}]
        set v [option get $menu v_$e {}]
        switch -- [option get $menu t_$e {}] {
            check {
                $menu add checkbutton -label $l -command $c \
                    -variable $v
            }
            radio {
                $menu add radiobutton -label $l -command $c \
                    -variable $v
            }
            separator {
                $menu add separator
            }
            cascade {
                set sub [option get $menu m_$e {}]
                if {[string length $sub] != 0} {
                    set submenu [menu $menu.$sub -tearoff 0]
                    $menu add cascade -label $l -command $c \
                        -menu $submenu
                    menubuttoninner $submenu
                }
            }
            default {
                $menu add command -label $l -command $c
            }
        }
    }
}
```

Managing User Preferences

This chapter describes a user preferences package. The X resource database is used to store preference settings. Applications specify what Tcl variables get initialized from what database entries. A user interface lets the user browse and change their settings.

User customization is an important part of any complex application. There are always design decisions that could go either way. A typical approach to choose a reasonable default but then allow users to change the default setting through a preferences user interface. This chapter describes a preference package that works by tying together a Tcl variable, which is used by the application, and an X resource specification, which can be set by the user. In addition, a user interface is provided so the user does not have to edit the resource database directly.

App-Defaults Files

We will assume that it is sufficient to have two sources of application defaults, a per-application database and a per-user database. In addition, we will allow for some resources to be specific to color and monochrome displays. The following example initializes the preference package by reading in the per-application and per-user resource specification files. There is also an initialization of the global array `pref` that will be used to hold state information about the preferences package. The `Pref_Init` procedure is called like this:

```
Pref_Init $library/foo-defaults ~/.foo-defaults
```

We assume `$library` is the directory holding support files for the `foo` application, and that per-user defaults will be kept `~/.foo-defaults`.

Example 28–1 Preferences initialization.

```
proc Pref_Init { userDefaults appDefaults } {
	global pref

	set pref(uid) 0;# for a unique identifier for widgets
	set pref(userDefaults) $userDefaults
	set pref(appDefaults) $appDefaults

	PrefReadFile $appDefaults startup
	if [file exists $userDefaults] {
		PrefReadFile $userDefaults user
	}
}
proc PrefReadFile { basename level } {
	if [catch {option readfile $basename $level} err] {
		Status "Error in $basename: $err"
	}
	if {[tk colormodel .] == "color"} {
		if [file exists $basename-color] {
			if [catch {option readfile \
					$basename-color $level} err] {
				Status "Error in $basename-color: $err"
			}
		}
	} else {
		if [file exists $basename-mono] {
			if [catch {option readfile $basename-mono $level
				Status "Error in $basename-mono: $err"
			}
		}
	}
}
```

The `PrefReadFile` procedure reads a resource file and then looks for another file with the suffix `-color` or `-mono` depending on the color model of the display . The `tk colormodel` command is used to find out what the toolkit thinks the display is capable of handling. The choices are either `color` or `monochrome`.

With this scheme a user would put generic settings in their `~/.foo-defaults` file, and they would put their color specifications in their `~/.foo-defaults-color` or `~/.foo-defaults-mono` files. Y ou could extend `PrefReadFile` to allow for per -host files as well.

Another approach is to use the `winfo visuals` command which provides more detailed information about the display characteristics. Y ou could detect a greyscale visual and support a third set of color possibilities. V isuals are discussed in Chapter 23.

Throughout this chapter we will assume that the `Status` procedure is used to display messages to the user . It could be as simple as:

```
proc Status { s } { puts stderr $s }
```

Defining Preferences

This section describes the `Pref_Add` procedure that is used by an application to define preference items. A preference item defines a relationship between a Tcl variable and an X resource name. A default value, a label, and a more extensive help string are also associated with the item. The Tcl variable is undefined at the time `Pref_Add` is called, then it is set from the value for the resource, if it exists, otherwise it is set to the default value. Each preference item will be represented by a Tcl list of these 5 elements. A few short routines hide the layout of the item lists and make the rest of the code read a bit better. `Pref_Add` is shown along with these below:

Example 28–2 Adding preference items.

```
proc PrefVar { item } { lindex $item 0 }
proc PrefXres { item } { lindex $item 1 }
proc PrefDefault { item } { lindex $item 2 }
proc PrefComment { item } { lindex $item 3 }
proc PrefHelp { item } { lindex $item 4 }

proc Pref_Add { prefs } {
    global pref
    append pref(items) $prefs
    foreach item $prefs {
        set varName [PrefVar $item]
        set xresName [PrefXres $item]
        set value [PrefValue $varName $xresName]
        if {$value == {}} {
            # Set variables that are still not set
            set default [PrefDefault $item]
            if {[llength $default] > 1} {
                if {[lindex $default 0] == "CHOICE"} {
                    PrefValueSet $varName [lindex $default 1]
                } else {
                    PrefValueSet $varName $default
                }
            } else {
                # Is it a boolean?
                if {$default == "OFF"} {
                    PrefValueSet $varName 0
                } elseif {$default == "ON"} {
                    PrefValueSet $varName 1
                } else {
                    # This is a string or numeric
                    PrefValueSet $varName $default
                }
            }
        } else {
            # Should map boolean resources to 0, 1 here.
        }
    }
}
```

(One small improvement can be made to `Pref_Add`. If a user specifies a boolean resource manually, they might use “true” instead of 1 and “false” instead of 0. `Pref_Add` should fix that up for us.)

The procedures `PrefValue` and `PrefValueSet` are used to query and set the value of the named variable, which can be an array element or a simple variable. The `upvar #0` command is used to set the variable in the global scope.

Example 28–3 Setting preference variables.

```
# PrefValue returns the value of the variable if it exists,
# otherwise it returns the X resource database value
proc PrefValue { varName xres } {
    upvar #0 $varName var
    if [info exists var] {
        return $var
    }
    set var [option get . $xres {}]
}
# PrefValueSet defines a variable in the globla scope.
proc PrefValueSet { varName value } {
    upvar #0 $varName var
    set var $value
}
```

An important side effect of the `Pref_Add` call is that the variables in the preference item are defined at the global scope. It is also worth noting that `PrefValue` will honor any existing value for a variable, so if the variable is already set at the global scope then neither the resource value or the default value will be used. It is easy to change `PrefValue` to always set the variable if this is not the behavior you want.

Example 28–4 Using the preferences package.

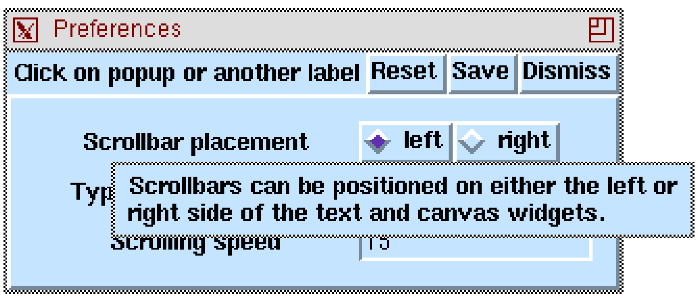
```
PrefAdd {
    {win(scrollside) scrollbarSide {CHOICE left right}
        “Scrollbar placement”
“Scrollbars can be positioned on either the left or
right side of the text and canvas widgets.”}
    {win(typeinkills) typeinKills OFF
        “Type-in kills selection”
“This setting determines whether or not the selection
is deleted when new text is typed in.”}
    {win(scrollspeed) scrollSpeed 15 “Scrolling speed”
“This parameter affects the scrolling rate when a selection
is dragged off the edge of the window. Smaller numbers
scroll faster, but can consume more CPU.”}
}
```

Any number of preference items can be specified in a call to `Pref_Add`. The list-of-lists structure is created by proper placement of the curly braces, and it is preserved when the argument is appended to the master list of preferences, pref(items). In this example Pref_Add gets passed a single argument that is a Tcl list with three elements. The Tcl variables are array elements, presumably related to the Win module of the application. The resource names are associated with the main application as opposed to any particular widget. They will be specified in the database like this:

```
*scrollbarSide: left
*typeinKills: 0
*scrollSpeed: 15
```

The Preferences User Interface

The figure shows what the interface looks like for the items added with the Pref_Add command given in the previous section. The popup window with the extended help text appears after you click on "Scrollbar placement".



The user interface to the preference settings is table driven. As a result of all the Pref_Add calls, a single list of all the preference items is built up. The interface is constructed by looping through this list and creating a user interface item for each

Example 28–5 A user interface to the preference items.

```
proc Pref_Dialog {} {
    global pref
    if [catch {toplevel .pref}] {
        raise .pref
    } else {
        wm title .pref "Preferences"
        set buttons [frame .pref.but]
        pack .pref.but -side top -fill x
        button $buttons.quit -text Quit \
            -command {PrefDismiss}
        button $buttons.save -text Save \
            -command {PrefSave}
        button $buttons.reset -text Reset \
            -command {PrefReset ; PrefDismiss}
```

```
        label $buttons.label \
            -text "Click labels for info on each item"
        pack $buttons.label -side left -fill x
        pack $buttons.quit $buttons.save $buttons.reset \
            -side right

        frame .pref.b -borderwidth 2 -relief raised
        pack .pref.b -fill both
        set body [frame .pref.b.b -bd 10]
        pack .pref.b.b -fill both

        set maxWidth 0
        foreach item $pref(items) {
            set len [string length [PrefComment $item]]
            if {$len > $maxWidth} {
                set maxWidth $len
            }
        }
        foreach item $pref(items) {
            PrefDialogItem $body $item $maxWidth
        }
    }
}
```

The interface supports three different types of preference items: boolean, choice, and general value. A boolean is implemented with a `checkbutton` that is tied to the Tcl variable, which will get a value of either 0 or 1. A boolean is identified by a default value that is either `ON` or `OFF`. A choice item is implemented as a set of `radiobuttons`, one for each choice. A choice item is identified by a default value that is a list with the first element equal to `CHOICE`. The remaining list items are the choices, with the first one being the default choice. If neither of these cases, boolean or choice, are detected, then an `entry` widget is created to hold the general value of the preference item.

Example 28–6 Interface objects for different preference types.

```
proc PrefDialogItem { frame item width } {
    global pref
    incr pref(uid)
    set f [frame $frame.p$pref(uid) -borderwidth 2]
    pack $f -fill x
    label $f.label -text [PrefComment $item] -width $width
    bind $f.label <1> \
        [list PrefItemHelp %X %Y [PrefHelp $item]]
    pack $f.label -side left
    set default [PrefDefault $item]
    if {([llength $default] > 1) &&
         ([lindex $default 0] == "CHOICE")} {
        foreach choice [lreplace $default 0 0] {
            incr pref(uid)
            radiobutton $f.c$pref(uid) -text $choice \
                -variable [PrefVar $item] -value $choice
```

```
            pack $f.c$pref(uid) -side left
        }
    } else {
        if {$default == "OFF" || $default == "ON"} {
            # This is a boolean
            set varName [PrefVar $item]
            checkbutton $f.check -text "On" -variable $varName
            pack $f.check -side left
        } else {
            # This is a string or numeric
            entry $f.entry -width 10 -relief sunken
            pack $f.entry -side left -fill x -expand true
            set pref(entry,[PrefVar $item]) $f.entry
            set varName [PrefVar $item]
            $f.entry insert 0 [uplevel #0 [list set $varName]]
            bind $f.entry <Return> "PrefEntrySet %W $varName"
        }
    }
}
proc PrefEntrySet { entry varName } {
    PrefValueSet $varName [$entry get]
}
```

The use of radio and check buttons that are tied directly to the Tcl variables results in a slightly different mode of operation for the preferences interface than is provided by other toolkits. Typically a user will make some settings and then choose `Save` or `Cancel`. In this interface, when the user clicks a `radiobutton` or a `checkbutton` then the Tcl variable is set immediately. Of course, there are still `Save` and `Cancel` buttons, but there is also an intermediate state in which the settings have been made but they have not been saved to a file. This is either a feature that lets users try out settings without committing to them, or it is a bug. However, changing this requires introducing a parallel set of variables to shadow the real variables until the user hits `Save`, which is tedious to implement.

In order to obtain a similar effect with the general preference item, the `<Return>` key is bound to a procedure that will set the associated Tcl variable to the value from the `entry` widget. `PrefEntrySet` is a one-line procedure that saves us from having to use the more awkward binding given below. Grouping with double-quotes allows substitution of $varName, but then we need to quote the square brackets to postpone command substitution.

```
bind $f.entry <Return> "PrefValueSet $varName \[%W get\]"
```

The binding on `<Return>` is done as opposed to using the `-textvariable` option because it interacts with traces on the variable a bit better. With `trace` you can arrange for a Tcl command to be executed when a variable is changed. For a general preference item it is better to wait until the complete value is entered before responding to its new value. A tracing example is given in the next section.

The other aspect of the user interface is the display of additional help information for each item. If there are lots of preference items then there isn't enough room to display this information directly. Instead, clicking on the short description for each item brings up a `toplevel` window with the help text for that item.

Example 28–7 Displaying the help text for an item.

```
proc PrefItemHelp { x y text } {
	catch {destroy .prefitemhelp}
	if {$text == {}} {
		return
	}
	set self [toplevel .prefitemhelp -class Itemhelp]
	wm title $self "Item help"
	wm geometry $self +[expr $x+10]+[expr $y+10]
	wm transient $self .pref
	message $self.msg -text $text -aspect 1500
	pack $self.msg
	bind $self.msg <1> {PrefNukeItemHelp .prefitemhelp}
	.pref.but.label configure -text \
		"Click on popup or another label"
}
proc PrefNukeItemHelp { t } {
	.pref.but.label configure -text \
		"Click labels for info on each item"
	destroy $t
}
```

Managing The Preferences File

The preference settings are saved in the per-user file. The file is divided into two parts. The tail is automatically re-written by the preferences package. Users can manually add resource specifications to the beginning of the file and they will be preserved.

Example 28–8 Saving preferences settings to a file.

```
# PrefSave writes the resource specifications to the
# end of the per-user resource file, allowing users to
# add other resources to the beginning.
proc PrefSave {} {
	global pref
	if [catch {
		set old [open $pref(userDefaults) r]
		set oldValues [split [read $old] \n]
		close $old
	}] {
		set oldValues {}
	}
	if [catch {open $pref(userDefaults).new w} out] {
		.pref.but.label configure -text \
		"Cannot save in $pref(userDefaults).new: $out"
		return
	}
```

```
	foreach line $oldValues {
		if {$line == \
				"!!! Lines below here automatically added"} {
			break
		} else {
			puts $out $line
		}
	}
	puts $out "!!! Lines below here automatically added"
	puts $out "!!! [exec date]"
	puts $out "!!! Do not edit below here"
	foreach item $preferences {
		set varName [PrefVar $item]
		set xresName [PrefXres $item]
		if [info exists pref(entry,$varName)] {
			PrefEntrySet $pref(entry,$varName) $varName
		}
		set value [PrefValue $varName $xresName]
		puts $out [format "%s\t%s" *${xresName}: $value]
	}
	close $out
	set new [glob $pref(userDefaults).new]
	set old [file root $new]
	if [catch {exec mv $new $old} err] {
		Status "Cannot install $new: $err"
		return
	}
	PrefDismiss
}
```

There is one fine point in PrefSave, which is that the value from the entry widget for general purpose items is obtained explicitly in case the user has not already pressed <Return> to update the Tcl variable.

Example 28–9 Read settings from the preferences file.

```
proc PrefReset {} {
	global pref
	# Re-read user defaults
	option clear
	PrefReadFile $pref(appDefaults) startup
	PrefReadFile $pref(userDefaults) user
	# Clear variables
	set items $pref(items)
	set pref(items) {}
	foreach item $items {
		uplevel #0 [list unset [PrefVar $item]]
	}
	# Restore values
	Pref_Add $items
}
proc PrefDismiss {} {
	destroy .pref
```

```
    catch {destroy .prefitemhelp}
}
```

The interface is rounded out with the PrefReset and PrefDismiss procedures. A reset is achieved by clearing the option database and reloading it, and then temporarily clearing the preference items and their associated variables and then redefining them with Pref_Add.

Tracing Changes To Preference Variables

Suppose, for example, we want to repack the scrollbars when the user changes their scrollside setting from left to right. This is done by setting a trace on the win(scrollside) variable. When the user changes that via the user interface, the trace routine will be called. The trace command and its associated procedure are given below. The variable must be declared global before setting up the trace, which is not otherwise required if Pref_Add is the only command using the variable.

Example 28–10 Tracing a Tcl variable in a preference item.

```
PrefAdd {
    {win(scrollside) scrollbarSide {CHOICE left right}
        "Scrollbar placement"
"Scrollbars can be positioned on either the left or
right side of the text and canvas widgets."}
}
global win
set win(lastscrollside) $win(scrollside)
trace variable win(scrollside) w ScrollFixup

# Assume win(scrollbar) identifies the scrollbar widget

proc ScrollFixup { name1 name2 op } {
    global win
    if {$win(scrollside) != $win(lastscrollside)} {
        set parent [lindex [pack info $win(scrollbar)] 1]
        pack forget $win(scrollbar)
        set firstchild [lindex [pack slaves $parent] 0]
        pack $win(scrollbar) -in $parent -before $firstchild \
            -side $win(scrollside) -fill y
        set win(lastscrollside) $win(scrollside)
    }
}
```

CHAPTER 29

C Programming and Tcl

This chapter explains how to extend the basic Tcl shells with new built-in commands. It describes how to include a Tcl interpreter in an existing application. The chapter reviews some of the support facilities provided by the Tcl C library, including a hash table package.

Tcl is designed to be easily extensible by writing new command implementations in C. A command implemented in C is more efficient than an equivalent Tcl procedure. A more pressing reason to write C code is that it may not be possible to provide the same functionality purely in Tcl. Suppose you have a new device, perhaps a color scanner or a unique input device. The programming interface to that device is through a set of C procedures that initialize and manipulate the state of the device. Without some work on your part, that interface is not accessible to your Tcl scripts. You are in the same situation if you have a C library that implements some specialized function such as a database. Foretunately, it is rather straight-forward to provide a Tcl interface that corresponds to the C interface. Unfortunately it is not automatic. This chapter explains how to provide a Tcl interface as a one or more new Tcl commands that you implement in C.

An alternative to writing new Tcl commands is to write stand-alone programs in C and use the Tcl `exec` command to run these programs. However, there is additional overhead in running an external program as compared to invoking a Tcl command that is part of the same application. There may be long lived state associated with your application (e.g., the database), and it may make sense for a collection of Tcl commands to provide an interface to this state than to run a program each time you want to access it. An external program is more suitable for one-shot operations like encrypting a file.

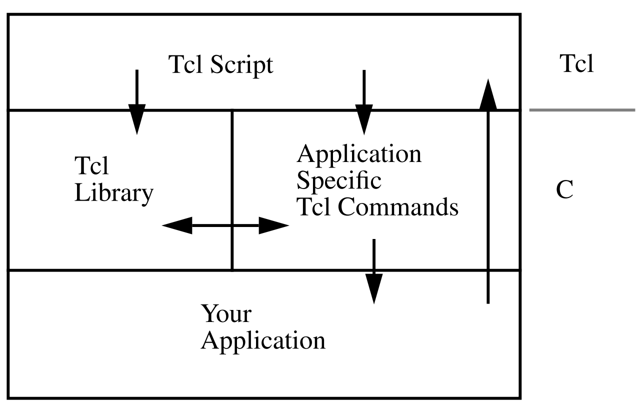
Another way to view Tcl is as a C library that is easy to integrate into your existing application. By adding the Tcl interpreter you can configure and control your application with Tcl scripts, and with Tk you can provide a nice graphical interface to it. This was the original model for Tcl. Applications would be largely application-specific C code and include a small amount of Tcl for configuration and the graphical interface. However, the basic Tcl shells proved so useful by themselves that relatively few Tcl programers need to worry about programming in C.

Using the Tcl C Library

This chapter does not provide a complete reference to the procedures exported by the Tcl C library. Instead, the general use of the procedures is explained at the end of this chapter, and a few of them appear in the code examples. You will need to refer to the on-line manual pages for the routines for the specific details about each procedure. This approach differs from the rest of the chapters on the Tcl scripting commands, but space and time preclude a detailed treatment of the Tcl C library. Besides, their man pages are an excellent source of information. The goal of this chapter is to give you an overall idea of what it is like to integrate C and Tcl, and to provide a few working examples.

Application Structure

This section describes the overall structure of an application that includes a Tcl interpreter. The relationship between the Tcl interpreter and the rest of your application can be set up in a variety of ways. A general picture is shown below.



The Tcl C library implements the interpreter and the core Tcl commands such as `set`, `while`, and `proc`. Application-specific Tcl commands are implemented in C or C++ and registered as commands in the interpreter. The interpreter calls these *command procedures* when the script uses the application-specific Tcl command. The command procedures are typically thin layers over existing functionality in your application. Finally, by using Tcl_Eval, your application can invoke functionality programmed in the script layer. You can query and set Tcl variables from C using the Tcl_SetVar and Tcl_GetVar procedures.

The application creates an interpreter with Tcl_CreateInterp and registers new commands with Tcl_CreateCommand. Then it evaluates a script to initialize the application by calling Tcl_EvalFile. The script can be as simple as defining a few variables that are parameters of a computation, or, it can be as ambitious as building a large user interface. The situation is slightly more complicated if you are using Tk and providing a graphical user interface, but not much more complex. Using Tk and C is described in the next chapter.

Tcl_Main and Tcl_AppInit

The Tcl library supports the basic application structure through the Tcl_Main procedure that is designed to be called from your main program. Tcl_Main does three things:

- It creates an interpreter that includes all the standard Tcl commands like set and proc. It also defines a few Tcl variables like argc and argv. These have the command line arguments that were passed to your application.
- It calls Tcl_AppInit, which is not part of the Tcl library. Instead, your application provides this procedure. In Tcl_AppInit you can register additional appcliation-specific Tcl commands.
- It reads a script or goes into an interactive loop.

To use Tcl_Main you call it from your main program and provide an implementation of the Tcl_AppInit procedure. An example is shown below.

Example 29–1 A canonical Tcl main program and Tcl_AppInit.

```
/* main.c */
#include <tcl.h>

/*
 * Declarations for application-specific command procedures
 */
int RandomCmd(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);

main(int argc, char *argv[]) {
   Tcl_Main(argc, argv);
   exit(0);
}
/*
 * Tcl_AppInit is called from Tcl_Main
 * after the Tcl interpreter has been created,
 * and before the script file
```

```
 * or interactive command loop is entered.
 */
int
Tcl_AppInit(Tcl_Interp *interp) {
   /*
    * Initialize packages
    * Tcl_Init sets up the Tcl library facility.
    */
   if (Tcl_Init(interp) == TCL_ERROR) {
      return TCL_ERROR;
   }
   /*
    * Register application-specific commands.
    */
   Tcl_CreateCommand(interp, "random", RandomCmd,
          (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
   /*
    * Define startup filename. This file is read in
    * case the program is run interactively.
    */
   tcl_RcFileName = "~/.myapp.tcl";
   return TCL_OK;
}
```

The `main` program calls `Tcl_Main` with the `argc` and `argv` parameters passed into the program. These are the strings passed to the program on the command line, and `Tcl_Main` will store these values into Tcl variables by the same name. `Tcl_AppInit` is called by `Tcl_Main` with one argument, a handle on a newly created interpreter. There are three parts to the `Tcl_AppInit` procedure. The first part initializes the various packages used by the application. The example just calls `Tcl_Init` to complete the setup of the Tcl interpreter. While the core Tcl commands are defined during `Tcl_CreateInterp`, which is called by `Tcl_Main`, there is some additional setup required for the Tcl library facility that is done by the `Tcl_Init` call. The library facility is described later in this chapter.

The second part of `Tcl_AppInit` does application-specific initialization. The example registers a command procedure, `RandomCmd` that implements a new Tcl command, `random`. When the Tcl script uses the `random` command, the `RandomCmd` procedure will be invoked by the Tcl library. The command procedure will be described in the next section. The third part defines an application startup script, `tcl_RcFileName`, that is used if the program is used interactively.

The standard main in Tcl 7.3

The `Tcl_Main` procedure was added in Tcl 7.4. Prior to that the Tcl library actually included a `main` program, so all you needed was a Tcl_AppInit procedure. There were some problems with using `main` from the library, especially with C++ programs, so `Tcl_Main` was introduced.

A C Command Procedure

The interface to a C command procedure is much like the interface to the `main` program. The arguments from the Tcl command are available as an array of strings defined by an `argv` parameter and counted by an `argc` parameter. In addition, the handle on the interpreter is passed, along with the client data that was registered when the command was defined. The client data is useful if the same command procedure is implementing many different commands. For example, a Tk command procedure can implement the commands corresponding to the various instances of a particular type of Tk widget. It that case, client data is used to hold a pointer to the data structure defining the particular instance of the widget.

The return value of a Tcl command is either a string or an error message. A field in the interp data structure is used to return this value, and the the command procedure returns either `TCL_OK` or `TCL_ERROR` to indicate success or failure. The procedure can also return `TCL_BREAK`, `TCL_CONTINUE`, or an application-specific code, which might be useful if you are implementing new kinds of built-in control structures. The examples in this book only use `TCL_OK` and `TCL_ERROR`. The use of the result field to return string values is described in the next section.

Example 29–2 The `RandomCmd` C command procedure.

```
/*
 * RandomCmd --
 * This implements the random Tcl command. With no arguments
 * the command returns a random integer.
 * With an integer valued argument "range",
 * it returns a random integer between 0 and range.
 */
int
RandomCmd(ClientData clientData, Tcl_Interp *interp,
		int argc, char *argv[])
{
	int rand, error;
	int limit = 0;
	if (argc > 2) {
		interp->result = "Usage: random ?range?";
		return TCL_ERROR;
	}
	if (argc == 2) {
		error = Tcl_GetInt(interp, argv[1], &limit);
		if (error != TCL_OK) {
			return error;
		}
	}
	rand = random();
	if (limit != 0) {
		rand = rand % limit;
	}
	sprintf(interp->result, "%d", rand);
	return TCL_OK;
```

```
}
```

The `random` implementation accepts an optional argument that is a range over which the random numbers should be returned. The `argc` parameter is tested to see if this argument has been given in the Tcl command. `argc` counts the command name as well as the arguments, so in our case `argc == 2` indicates that the command has been invoked something like:

```
random 25
```

The procedure `Tcl_GetInt` is used to convert the string-valued argument to an integer. It does error checking and sets the interpreter's result field in the case of error, so we can just return if it fails to return `TCL_OK`.

Finally, the real work of calling `random` is done, and the result is formatted directly into the result buffer. `TCL_OK` is returned to signal success.

Managing The Result's Storage

There is a simple protocol used to manage the storage for a command procedure's result string. It involves two fields in the interpreter structure, `interp->result` that holds the value, and `interp->freeProc` that determines how the storage is cleaned up. When a command is called the interpreter provides default storage of `TCL_RESULT_SIZE` bytes, which is 200 by default. The default cleanup action is to do nothing. These defaults support two simple ways to define the result of a command. One way is to use `sprintf` to format the result in place.

```
sprintf(interp->result, "%d", rand);
```

Using sprintf is suitable if you know your result string is short, which is often the case. The other way is to set `interp->result` to the address of a constant string. In this case the original result buffer is not used, and there is no cleanup required because the string is compiled into the program.

```
interp->result = "Usage: random ?random?";
```

In more general cases the following procedures should be used to manage the `result` and `freeProc` fields.

```
Tcl_SetResult(interp, string, freeProc)
Tcl_AppendResult(interp, str1, str2, str3, (char *)NULL)
Tcl_AppendElement(interp, string)
```

`Tcl_SetResult` sets the return value to be *`string`*. The *`freeProc`* argument describes how the result should be disposed of. `TCL_STATIC` is used in the case where the result is a constant string allocated by the compiler. `TCL_DYNAMIC` is used if the result is allocated with `malloc`. `TCL_VOLATILE` is used if the result is in a stack variable. In this case the Tcl interpreter will make a copy of the result before calling any other command procedures. Finally, if you have your own memory allocator, pass in the address of the procedure that should free up the result.

`Tcl_AppendResult` copies its arguments into the result buffer, reallocating the buffer if necessary. The arguments are concatenated onto the end of the existing result, if any. `Tcl_AppendResult` can be called several times in order to build up a result.

`Tcl_AppendElement` adds the string to the result as a proper Tcl list element. It may add braces or backslashes to get the proper structure.

Invoking Scripts From C

The main program is not the only place you can evaluate a Tcl script. The `Tcl_Eval` procedure can be used essentially at any time to evaluate a Tcl command.

```
Tcl_Eval(Tcl_Interp *interp, char *command);
```

This is how the command associated with a button is invoked, for example. The only caveat about this is that the script may destroy the widget or Tcl command that invoked it. To guard against this, the `Tk_Preserve`, `Tk_Release`, and `Tk_EventuallyFree` procedures can be used to manage any data structures associated with the widget or Tcl command. These are described on page 363.

You should also be aware that `Tcl_Eval` may modify the string that is passed into it as a side effect of the way substitutions are performed. If you pass a constant string to `Tcl_Eval`, make sure your compiler hasn't put the string constant into read-only memory. If you use the *gcc* compiler you may need to use the `-fwritable-strings` option.

Bypassing Tcl_Eval

In a performance critical situation you may want to avoid some of the overhead associated with `Tcl_Eval`. David Nichols showed me a clever trick by which you can call the implementation of a C command procedure directly. The trick is facilitated by the `Tcl_GetCommandInfo` procedure that returns the address of the C command procedure for a Tcl command, plus its client data pointer. The `Tcl_Invoke` procedure shown in the next example implements this trick. It is used much like `Tcl_VarEval`, except that each of its arguments becomes an argument to the Tcl command without any substitutions being performed.

For example, you might want to insert a large chunk of text into a text widget without worriying about the parsing done by `Tcl_Eval`. You could use `Tcl_Invoke` like this:

```
Tcl_Invoke(interp, ".t", "insert", "insert", buf, NULL);
```

Or:

```
Tcl_Invoke(interp, "set", "foo", "$xyz [blah] {", NULL);
```

No substitutions are performed on any of the arguments because `Tcl_Eval` is out of the picture. The variable `foo` gets the literal value `$xyz [blah] {`.

Example 29–3 Calling C command procedure directly.

```
#include <varargs.h>
#include <tcl.h>
/*
 * Tcl_Invoke --
 *  Call this somewhat like Tcl_VarEval:
 * Tcl_Invoke(interp, cmdName, arg1, arg2, ..., NULL);
 * Each arg becomes one argument to the command,
 * with no substitutions or parsing.
 */
int
Tcl_Invoke(va_alist)
    va_dcl          /* Variable number of arguments */
{
    Tcl_Interp *interp;
    char *cmd;
    char **argv;
    int argc, max;
    Tcl_CmdInfo info;
    va_list pvar;
    int result;

    va_start(pvar);
    interp = va_arg(pvar, Tcl_Interp *);
    cmd = va_arg(pvar, char *);
    /*
     * Build an argv vector out of the rest of the arguments.
     */
    max = 10;
    argv = (char **)malloc(max * sizeof(char *));
    argv[0] = cmd;
    argc = 1;
    while (1) {
        argv[argc] = va_arg(pvar, char *);
        if (argv[argc] == (char *)NULL) {
            break;
        }
        argc++;
        if (argc >= max) {
            /*
             * Allocate a bigger vector and copy old values in.
             */
            int i;
            char **oldargv = argv;
            argv = (char **)malloc(2*max * sizeof(char *));
            for (i=0 ; i<max ; i++) {
                argv[i] = oldargv[i];
            }
            free(oldargv);
            max = 2*max;
        }
    }
    Tcl_ResetResult(interp);
    /*
```

```
     * Map from the command name to a C procedure.
     */
    if (Tcl_GetCommandInfo(interp, cmd, &info)) {
       result = (*info.proc)(info.clientData, interp,
             argc, argv);
    } else {
       Tcl_AppendResult(interp, "Unknown command \"",
          cmd, "\"", NULL);
       result = TCL_ERROR;
    }
    va_end(pvar);
    free(argv);
    return result;
}
```

Putting A Tcl Program Together

Assuming you've put the examples into files named tclMain.c, random.c, and tclInvoke.o you are ready to try them out. You need to know the locations of two things, the tcl.h include file and the tcl C library. In this book we'll assume they are in /usr/local/include and /usr/local/lib, respectively, but you should check with your local system administrator to see how things are set up at your site.

Example 29–4 A Makefile for a simple Tcl C program.

```
INC = -I/usr/local/include
LIBS = -L/usr/local/lib -ltcl -lm
DEBUG = -g
CFLAGS =$(DEBUG) $(INC)

OBJS = tclMain.o random.o tclInvoke.o

mytcl : $(OBJS)
       $(CC) -o mytcl $(OBJS) $(LIBS)
```

The details in this Makefile may not be correct for your system. In some cases the math library (-lm) is included in the standard C library. You should consult a local expert and define a Makefile so you can record the details specific to your site.

An Overview of the Tcl C library

This section provides a brief survey of other facilities provided by the Tcl C library. For the complete details about each procedure mentioned here, consult the on-line manual pages. The man pages describe groups of related C procedures. For example, Tcl_CreateCommand and Tcl_DeleteCommand are described in the CrtCommand man page. Your site may not have additional links setup to let you utter "man Tcl_CreateCommand". Instead, you may have to use "man CrtCommand". For this reason, the name of the man page is noted in each section that introduces the procedures.

Application initialization

The Tcl_Main and Tcl_AppInit procedures are described in the AppInit and Tcl_Main man pages, respectively.

Creating and deleting interpreters

A Tcl interpreter is created and deleted with the Tcl_CreateInterp and Tcl_DeleteInterp procedures, which are described in the CrtInterp man page. You can register a callback to occur when the interpreter is deleted with Tcl_CallWhenDeleted. Unregister the callback with Tcl_DontCallWhenDeleted. These two procedures are described in the CallDel man page.

Creating and deleteing commands

Register a new Tcl command with Tcl_CreateCommand, and delete a command with Tcl_DeleteCommand. The Tcl_GetCommandInfo and Tcl_SetCommandInfo procedures query and modify the procedure that implement a Tcl command and the clientdata that is associated with the command. All of these are described in the CrtCommand man page.

Managing the result string

The result string is managed through the Tcl_SetResult, Tcl_AppendResult, Tcl_AppendElement, and Tcl_ResetResult procedures. These are described in the SetResult man page. Error information is managed with the Tcl_AddErrorInfo, Tcl_SetErrorCode, and Tcl_PosixError procedures, which are described in the AddErrInfo man page.

Lists and command parsing

If you are reading commands, you can test for a complete command with Tcl_CommandComplete, which is described in the CmdCmplt man page. You can do backslash substitutions with Tcl_Backslash, which is described in the Backslash man page. The Tcl_Concat procedure, which is described in the Concat man page, concatenates its arguments with a space separator, just like the Tcl concat command.

You can chop a list up into its elements with Tcl_SplitList, which returns an array of strings. You can create a list out of an array of strings with Tcl_Merge. This behaves like the list command in that it will add syntax to the strings so that the list structure has one element for each of the strings. The

Tcl_ScanElement and Tcl_ConvertElement procedures are used by Tcl_Merge. All of these are described in the SplitList man page.

Command pipelines

The Tcl_CreatePipeline procedure does all the work of setting up a pipeline between processes. It handles file redirection and implements all the syntax supported by the exec and open commands. It is described by the CrtPipelin man page.

If the command pipeline is run in the background, then a list of process identifiers is returned. You can detach these processes with Tcl_DetachPids, and you can clean up after them with Tcl_ReapDetachedProcs. These are described in the DetachPid man page.

Tracing the actions of the Tcl interpreter

There are several procedures that let you trace the execution of the Tcl interpreter and provide control over its behavior. The Tcl_CreateTrace registers a procedure that is called before the execution of each Tcl command. Remove the registration with Tcl_DeleteTrace. These are described in the CrtTrace man page.

You can trace modifications and accesses to Tcl variables with Tcl_TraceVar and Tcl_TraceVar2. The second form is used with array elements. Remove the traces with Tcl_UntraceVar and Tcl_UntraceVar2. You can query the traces on variables with Tcl_VarTraceInfo and Tcl_VarTraceInfo2. These are all described in the TraceVar man page.

Evalutating Tcl commands

The Tcl_Eval command is used to evaluate a string as a Tcl command. Tcl_VarEval takes a variable number of string arguments and concatenates them before evaluation. The Tcl_EvalFile command reads commands from a file. Tcl_GlobalEval evaulates a string at the global scope. These are all described in the Eval man page.

If you are implementing an interactive command interpreter and want to use the history facility, then call Tcl_RecordAndEval. This records the command on the history list and then behaves like Tcl_GlobalEval. This is described in the RecordEval man page.

You can set the recursion limit of the interpreter with Tcl_SetRecursionLimit, which is described in the SetRecLmt man page.

If you are implementing a new control structure you may need to use the Tcl_AllowExceptions procedure. This makes it OK for Tcl_Eval and friends to return something other than TCL_OK and TCL_ERROR. This is described in the AllowExc man page.

Manipulating Tcl variables

You can set a Tcl variable with `Tcl_SetVar` and `Tcl_SetVar2`. The second form is used for array elements. You can retrieve the value of a Tcl variable with `Tcl_GetVar` and `Tcl_GetVar2`. You can delete variables with `Tcl_UnsetVar` and `Tcl_UnsetVar2`. These are all described in the `SetVar` man page.

You can link a Tcl variable and a C variable together with `Tcl_LinkVar`, and break the relationship with `Tcl_UnlinkVar`. Setting the Tcl variable modifies the C variable, and reading the Tcl variable returns the value of the C variable. These are described in the `LinkVar` man page.

Use the `Tcl_UpVar` and `Tcl_UpVar2` procedures to link Tcl variables from different scopes together. You may need to do if your command takes the name of a variable as an argument as opposed to a value. These procedures are used in the implementation of the `upvar` Tcl command, and they are described in the `UpVar` man page.

Evalutating expressions

The Tcl expression evaluator is available through the `Tcl_ExprLong`, `Tcl_ExprDouble`, `Tcl_ExprBool` and `Tcl_ExprString` procedures. These all use the same evaluator, but they differ in how they return their result. They are described in the `ExprLong` man page.

You can register the implementation of new math functions by using the `Tcl_CreateMathFunc` procedure, which is described in the `CrtMathFnc` man page.

Converting numbers

You can convert strings into numbers with the `Tcl_GetInt`, `Tcl_GetDouble`, and `Tcl_GetBoolean` procedures, which are described in the `GetInt` man page. The `Tcl_PrintDouble` procedure converts a floating point number to a string. It is used by Tcl anytime it need to do this conversion, and it honors the precision specified by the `tcl_precision` variable. It is described in the `PrintDbl` man page.

Hash tables

Tcl has a nice hash table package that automatically grows the hash table data structures as more elements are added to the table. Because everything is a string, you may need to set up a hash table that maps from a string-valued key to an internal data structure. The procedures in the package are `Tcl_InitHashTable`, `Tcl_DeleteHashTable`, `Tcl_CreateHashEntry`, `Tcl_DeleteHashEntry`, `Tcl_FindHashEntry`, `Tcl_GetHashValue`, `Tcl_SetHashValue`, `Tcl_GetHashKey`, `Tcl_FirstHashEntry`, `Tcl_NextHashEntry`, and `Tcl_HashStats`. These are described in the `Hash` man page.

Dynamic Strings

The Tcl dynamic string package is designed for strings that get built up incrementaly. You will need to use dynamic strings if you use the `Tcl_TildeSubst` procedure. The procedures in the package are `Tcl_DStringInit`, `Tcl_DStringAppend`, `Tcl_DStringAppendElement`, `Tcl_DStringStartSublist`, `Tcl_DStringEndSublist`, `Tcl_DStringLength`, `Tcl_DStringValue`, `Tcl_DStringSetLength`, `Tcl_DStringFree`, `Tcl_DStringResult`, and `Tcl_DStringGetResult`. These are described in the `DString` man page.

Regular expressions and string matching

The regular expression library used by Tcl is exported through the `Tcl_RegExpMatch`, `Tcl_RegExpCompile`, `Tcl_RegExpExec`, and `Tcl_RegExpRange` procedures. These are described in the `RegExp` man page. The string match function is available through the `Tcl_StringMatch` procedure, which is described in the `StrMatch` man page.

Tilde Substitution

The `Tcl_TildeSubst` procedure converts filenames that begin with ~ into absolute pathnames. The ~ syntax is used to refer to the home directory of a user.

Working with signals

Tcl provides a simple package for safely dealing with signals and other asynchronous events. You register a handler for an event with `Tcl_AsyncCreate`. When the event occurs, you mark the handler as ready with `Tcl_AsyncMark`. When the Tcl interpreter is at a safe point, it uses `Tcl_AsyncInvoke` to call all the ready handlers. Your application can call `Tcl_AsyncInvoke`, too. Use `Tcl_AsyncDelete` to unregister a handler. These are described in the `Async` man page.

CHAPTER 30

C Programming and Tk

This chapter explains how to include Tk in your application. It includes an overview of the Tk C library. The next chapter shows a sample widget implementation.

Tk has a few ways of its own that it can be extended. You can implement new widgets, new canvas items, new image types, and new geometry managers. This chapter provides a brief introduction to these topics and some examples. Geometry managers are not described, although you can read about the table geometry manager provided by the BLT extension in the next chapter.

The structure of an application that uses Tk is a little different than the basic structure outlined in the previous chapter. After an initialization phase your program enters an event loop so it can process window system events. If you use certain extensions like Tcl-DP, you will also need an event loop. `Tk_MainLoop` is an event loop that processes window events, and `Tk_DoOneEvent` can be used if you build your own event loop. If you use `Tk_MainLoop`, you can have it call handlers for your own I/O streams by using `Tk_CreateFileHandler`. Thus there is some initial setup, the evaluation of a script, and then a processing loop.

Tk_Main and Tcl_AppInit

The Tk library supports the basic application structure through the `Tk_Main` procedure that is designed to be called from your `main` program. `Tk_Main` does the following things:

- Like `Tcl_Main` it creates a Tcl interpreter and defines the `argc` and `argv` Tcl variables. The complete set of variables is listed below .

- It parses some window-related command line arguments. These are listed below.
- It creates the main window for your application by calling Tk_CreateMainWindow. It also defines the env(DISPLAY) variable.
- It calls Tcl_AppInit, which is is provided by your application. Y our Tcl_AppInit should call Tcl_Init and Tk_Init as shown in the example.
- It reads a script or sets up to read interactive commands.
- It enters an event loop in order to process window events and interactive commands.

Example 30–1 A canonical Tk main program and Tcl_AppInit.

```
/* main.c */
#include <tk.h>

main(int argc, char *argv[]) {
    Tk_Main(argc, argv);
    exit(0);
}
/*
 * New features added by this wish.
 */
int ClockCmd(ClientData clientData,
             Tcl_Interp *interp,
             int argc, char *argv[]);

/*
 * Tcl_AppInit is called from Tcl_Main
 * after the Tcl interpreter has been created,
 * and before the script file
 * or interactive command loop is entered.
 */
int
Tcl_AppInit(Tcl_Interp *interp) {
    /*
     * Initialize packages
     * Tcl_Init sets up the Tcl library facility.
     */
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    /*
     * Define application-specific commands here.
     */
    Tcl_CreateCommand(interp, "clock", ClockCmd,
        (ClientData)Tk_MainWindow(interp),
        (Tcl_CmdDeleteProc *)NULL);
```

```
    /*
     * Define startup filename. This file is read in
     * case the program is run interactively.
     */
    tcl_RcFileName = "~/.myapp.tcl";
    return TCL_OK;
}
```

The use of Tk_Main is very similar to using Tcl_Main. Both procedures call Tcl_AppInit for initialization. If you are using Tk then you need to call both Tcl_Init and Tk_Init from your Tcl_AppInit procedure. The first sets up the Tcl library, and the second sets up the script library used with the Tk widgets. This is important because much of the default behavior and event bindings for the Tk widgets are defined by its script library.

This example sets up for the clock widget example, the pixmap image type, and the label canvas item type that are the subject of examples later in this chapter.

A Custom Main Program

In more complex applications you may need to have complete control over the main program. This section gives an example that has a custom event loop. It shows much of the boiler-plate code needed to initialize a Tk application. In addition, it sets up an error handler for X protocol errors. This is mainly useful so you can set a breakpoint and find out what is causing the problem.

You should carefully consider whether a custom main program is really necessary. The primary point of this example is to give you an understanding of what goes on inside Tk_Main. In most cases Tk_Main should be sufficient for your needs.

Example 30–2 A custom Tk main program.

```
#include <tk.h>
/*
 * XErrorProc --
 *     Toe-hold for debugging X Protocol botches.
 */
static int
XErrorProc(data, errEventPtr)
    ClientData data;
    XErrorEvent *errEventPtr;
{
    Tk_Window w = (Tk_Window)data;
    fprintf(stderr, "X protocol error: ");
    fprintf(stderr, "error=%d request=%d minor=%d\n",
        errEventPtr->error_code, errEventPtr->request_code,
        errEventPtr->minor_code);
    /*
```

```
        * Claim to have handled the error.
        */
       return 0;
}

Tk_Window mainWindow;

/*
 * A table for command line arguments.
 */
static char *display = NULL;
static int debug = 0;
static char *geometry = NULL;

Tk_ArgvInfo argTable[] = {
    {"-display", TK_ARGV_STRING, (char *) NULL,
    (char *) &display, "Display to use"},
    {"-debug", TK_ARGV_CONSTANT, (char *) 1, (char *) &debug,
       "Set things up for gdb-type debugging"},
    {"", TK_ARGV_END, },
};
/*
 * This program takes one argument, which is the
 * name of a script to interpret.
 */
main(int argc, char *argv[])
{
    Tcl_Interp *interp;
    int error; char *trace;

    interp = Tcl_CreateInterp();
    if (Tk_ParseArgv(interp, (Tk_Window) NULL, &argc, argv,
       argTable, 0) != TCL_OK) {
       fprintf(stderr, "%s\n", interp->result);
       exit(1);
    }
    if (argc < 2) {
       fprintf(stderr, "Usage: %s filename\n", argv[0]);
       exit(1);
    }

    /*
     * Create the main window. The name of the application
     * for use with the send command is "myapp". The
     * class of the application for X resources is "Myapp".
     */
    mainWindow = Tk_CreateMainWindow(interp, display,
       "myapp", "Myapp");
    if (mainWindow == NULL) {
       fprintf(stderr, "%s\n", interp->result);
       exit(1);
    }
    /*
     * Register the X protocol error handler, and ask for
     * a synchronous protocol to help debugging.
```

```
     */
    Tk_CreateErrorHandler(Tk_Display(mainWindow), -1, -1, -1,
       XErrorProc, (ClientData)mainWindow);
    if (debug) {
       XSynchronize(Tk_Display(mainWindow), True);
    }
    /*
     * Grab an initial size and background.
     */
    Tk_GeometryRequest(mainWindow, 200, 200);
    Tk_SetWindowBackground(mainWindow,
       WhitePixelOfScreen(Tk_Screen(mainWindow)));

    /*
     * This is where Tcl_AppInit would be called.
     * In this case, we do the work right here.
     */
    if (Tcl_Init(interp) != TCL_OK) {
       fprintf(stderr, "Tcl_Init failed: %s\n",
            interp->result);
    }
    if (Tk_Init(interp) != TCL_OK) {
       fprintf(stderr, "Tk_Init failed: %s\n",
            interp->result);
    }
    error = Tcl_EvalFile(interp, argv[1]);
    if (error != TCL_OK) {
       fprintf(stderr, "%s: %s\n", argv[1],
              interp->result);
       trace = Tcl_GetVar(interp, "errorInfo",
              TCL_GLOBAL_ONLY);
       if (trace != NULL) {
          fprintf(stderr, "*** TCL TRACE ***\n");
          fprintf(stderr, "%s\n", trace);
       }
    }
    /*
     * Enter the custom event loop.
     */
    while (MyappExists()) {
       Tk_DoOneEvent(TK_ALL_EVENTS);
       MyappStuff();
    }
    /*
     * Call the Tcl exit to ensure that everything is
     * cleaned up properly.
     */
    Tcl_Eval(interp, "exit");
    return 0;
}
```

The command line arguments are parsed with `Tk_ParseArgv`. Then `Tcl_CreateInterp` creates an interpreter context, and `Tk_CreateMainWindow` creates the first window. As a side effect it defines all the Tk-related Tcl commands. The default window size is set, and the rest of the appearance is left up to the script. `Tcl_Init` and `Tk_Init` are called to complete the setup of these packages. `Tk_Init` has to be called after `Tk_CreateMainWindow`.

The handler for X protocol errors is installed with `Tk_CreateErrorHandler`. If the `debug` flag is set, then the X protocol is put into synchronous mode. This means that any protocol errors will occur as a direct result of your graphic operations, so you can put a breakpoint in `XErrorProc` and see what call causes the problems.

The application is really defined by the script, which is processed by the `Tcl_EvalFile` command. Its file name argument is `argv[1]`, which is the first argument to the program when it is run from the command line. If the user types a bad file name, then `Tcl_EvalFile` will return an error so we can avoid checking for that ourselves.

This argument convention means that you can specify your program directly in the script with the `#!` notation. That is, if your program is named `myapp`, and it is stored as `/usr/joe/bin/myapp`, then you can begin a script with:

```
#!/usr/joe/bin/myapp
```

The script will be processed by your version of the Tcl interpreter Remember there is a 32 character limit on this line in most UNIX systems, including the `#!`.

Much of the main program is devoted to handling any errors from the script. First, the return code from `Tcl_EvalFile` is checked. If it is not `TCL_OK`, then an error has occurred in the script. An error message is available in `interp->result`. We can provide even more detailed information to the user than the error message generated by the offending command. The interpreter maintains a variable `errorInfo` that is a stack trace of the commands that led up to the error. The `Tcl_GetVar` call returns us its value, or `NULL` if it is undefined. In practice, you would probably prompt the user before dumping the Tcl trace.

A Custom Event Loop

An event loop is used to process window system events and other events like timers and network sockets. The different event types are described below. All Tk applications must have an event loop so they function properly in the window system environment. Tk provides a standard event loop with the `Tk_MainLoop` procedure, which should be sufficient for most cases.

You can provide your own event loop as shown in the previous example. In this case you call `Tk_DoOneEvent` to process any outstanding Tk events. By default, `Tk_DoOneEvent` handles all event types and will block if there are no events ready. It takes a bitmap of flag arguments that control what kind of events it will handle and whether or not it will block. Specify the `TK_DONT_WAIT` flag if you don't want it to block. In this case you typically want to process all outstanding requests and then go do some application-specific processing. `Tk_DoOneEvent` returns 1 if there are more events ready to process.

Example 30–3 Using `Tk_DoOneEvent` with `TK_DONT_WAIT`.

```
void
DoAllTkEvents() {
    while (Tk_DoOneEvent(TK_ALL_EVENTS|TK_DONT_WAIT)) {
        /* keep processing Tk events */
    }
}
```

The other way to customize the event loop is to register handlers for different events and use the `Tk_MainLoop` procedure. `Tk_MainLoop` takes no parameters and it returns when the last window is destroyed. It uses `Tk_DoOneEvent` to process events. Unless you have some really special requirements, using `Tk_MainLoop` and the registration procedures described below is preferable to using `Tk_DoOneEvent` directly .

There are four event classes, and they are handled in the following order by `Tk_DoOneEvent`.

- Window events. Use the `Tk_CreateEventHandler` procedure to register a handler for these events. Use the `TK_X_EVENTS` flag to process these in `Tk_DoOneEvent`.
- File events. Use these events to wait on slow devices and network connections. Register a handler with `Tk_CreateFileHandler`. Use the `TK_FILE_EVENTS` flag to process these in `Tk_DoOneEvent`.
- Timer events. You can set up events to occur after a specified time period. Use the `Tk_CreateTimerHandler` procedure to register a handler for the event. Use the `TK_TIMER_EVENTS` flag to process these in `Tk_DoOneEvent`.
- Idle events. These events are processed when there is nothing else to do. Virtually all the Tk widgets use idle events to display themselves. Use the `Tk_DoWhenIdle` procedure to register a procedure to call once at the next idle time. Use the `TK_IDLE_EVENTS` flag to process these in `Tk_DoOneEvent`.

An Overview of the Tk C library.

The next few sections briefly introduce the facilities provided by the Tk C library. For the complete details you will need to consult the on line manual pages. The man page for each set of routines is identified in the description so you can easily find the right on-line documentation. Your site may not be set up so that the man page is available by the name of the routine. You many need to know the name of the man page first.

Parsing command line arguments

The `Tk_ParseArgv` procedure parses command line arguments. This procedure is designed for use by main programs. While you could use it for Tcl commands, the Tk_ConfigureWidget procedure might be better suited. The Tk_ParseArgv procedure is described by the ParseArgv man page.

The standard application setup

The Tk_Main procedure does the standard setup for your application's main window and event loop. It is described by the Tk_Main man page.

Creating windows

The Tk_CreateMainWindow procedure is what Tk_Main uses to create the main window for your application. The Tk_CreateWindow and Tk_CreateWindowFromPath are used to create windows for widgets. The actual creation of the window in the X server is delayed until an idle point. You can force the window to be created with Tk_MakeWindowExist. Destroy a window with Tk_DestroyWindow. These are described in the CrtMainWin man page.

The Tk_MainWindow procedure returns the handle on the applications main window. It is described in the MainWin man page. The Tk_MapWindow and Tk_UnmapWindow are used to display and withdraw a window, respectively. They are described in the MapWindow man page. The Tk_MoveToplevelWindow call is used to position a toplevel window. It is described in the MoveToplev man page.

Translate between window names and the Tk_Window type with Tk_Name, Tk_PathName, and Tk_NameToWindow. These are described in the Name man page.

Application name for send

The name of the application is defined or changed with Tk_SetAppName. This name is used when other applications send it Tcl commands using the send command. This procedure is described in the SetAppName man page.

Configuring windows

The configuration of a window includes its width, height, cursor, and so on. Tk provides a set of routines that use Xlib routines to configure a window and also cache the results. This makes it efficient to query these settings because the X server does not need to be contacted. The window configuration routines are Tk_ConfigureWindow, Tk_MoveWindow, Tk_ResizeWindow, Tk_MoveResizeWindow, Tk_SetWindowBorderWidth Tk_ChangeWindowAttributes, Tk_SetWindowBackground, Tk_SetWindowBackgroundPixmap, Tk_SetWindowBorder, Tk_SetWindowBorderPixmap, Tk_SetWindowColormap, Tk_DefineCursor, and Tk_UndefineCursor. These are described in the ConfigWind man page.

Window coordinates

The coordinates of a widget relative to the root window (the main screen) are returned by Tk_GetRootCoords. This is described in the GetRootCrd man page. The `Tk_GetVRootGeometry` procedure returns the size and position of a window relative to the virtual root window . This is described by the `GetVRoot` man page. The `Tk_CoordsToWindow` procedure locates the window under a given coordinate. It is described in the `CoordToWin` man page.

Window stacking order

Control the stacking order of windows in the X window hierarchy with `Tk_RestackWindow`. This is described in the `Restack` man page.

Window information

Tk keeps lots of information associated with each window , or wiget. The following calls are fast macros that return the information without calling the X server: `Tk_WindowId`, `Tk_Parent`, `Tk_Display`, `Tk_DisplayName`, `Tk_ScreenNumber`, `Tk_Screen`, `Tk_X`, `Tk_Y`, `Tk_Width`, `Tk_Height`, `Tk_Changes`, `Tk_Attributes`, `Tk_IsMapped`, `Tk_IsTopLevel`, `Tk_ReqWidth`, `Tk_ReqHeight`, `Tk_InternalBorderWidth`, `Tk_Visual`, `Tk_Depth`, and `Tk_Colormap`. These are described in the `WindowId` man page.

Configuring widget attributes

The `Tk_WidgetConfigure` procedure parses command line specification of attributes and allocates resources like colors and fonts. Related procedures include `Tk_Offset`, `Tk_ConfigureInfo`, `Tk_ConfigureValue`, `Tk_FreeOptions`, and these are described in the `ConfigWidg` man page.

Safe handling of the widget data structure

If your widget makes callbacks to the script level it might invoke a Tcl command that deletes the widget. To avoid havoc in such situations, a simple reference counting scheme can be implemented for data structures. Call `Tk_Preserve` to increment the use count, and call `Tk_Release` to decrement the count. Then, when your widget is destroyed, use the `Tk_EventuallyFree` procedure to indirectly call the procedure that cleans up your widget data structure. If the data structure is in use, then the clean up call is delayed until after the last reference to the data structure is released with `Tk_Release`. These procedures are described in the `Preserve` man page.

The selection and clipboard

Retrieve the current selection with `Tk_GetSelection`. This is described in the `GetSelect` man page. Register a handler for X selection requests with `Tk_CreateSelHandler`. Unregister the handler with `Tk_DeleteSelHandler`. These are described in the `CrtSelHdlr` man page. Claim ownership of the selection with `Tk_OwnSelection`. This is described in the `OwnSelect` man page.

Manipulate the clipboard with `Tk_ClipboardClear` and `Tk_ClipboardAppend`, which are described in the `Clipboard` man page.

Event bindings

The routines that manage bindings are exported by the Tk library so you can manage bindings your self. For example, the canvas widget does this to implement bindings on canvas items. The procedures are `Tk_CreateBindingTable`, `Tk_DeleteBindingTable`, `Tk_CreateBinding`, `Tk_DeleteBinding`, `Tk_GetBinding`, `Tk_GetAllBindings`, `Tk_DeleteAllBindings`, and `Tk_BindEvent`. These are described in the `BindTable` man page.

Event loop interface

The standard event loop is implemented by `Tk_MainLoop`. If you write your own event loop you need to call `Tk_DoOneEvent` so Tk can handle its events. If you read X events directly, e.g., through `Tk_CreateGenericHandler`, then you can dispatch to the correct handler for the event with `Tk_HandleEvent`. These are described in the `DoOneEvent` man page.

If you want to use the Tk event loop mechnaims without using the rest of Tk tookit, which requires a connection to an X server, then call `Tk_EventInit` to set up the event registration mechanism. You can create handlers for file, timer, and idle events after this call.

Restrict or delay events with the `Tk_RestrictEvent` procedure, which is described in the `RestrictEv` man page.

Handling X events

Use `Tk_CreateEventHandler` to set up a handler for specific X events. Widget implementations need a handler for expose and resize events, for example. Remove the registration with `Tk_DeleteEventHandler`. These are described in the `EventHndlr` man page.

You can set up a handler for all X events with `Tk_CreateGenericHandler`. This is useful in some modal interactions where you have to poll for a certain event. Delete the handler with `Tk_DeleteGenericHandler`. These are described in the `CrtGenHdlr` man page.

File handlers

Use `Tk_CreateFileHandler` to register handlers for I/O streams. You set up the handlers to be called when the I/O stream is ready for reading or writing, or both. Or, you can use the lower-level `Tk_CreateFileHandler2`, which is called every time through the event loop so it can decide for itself if the I/O stream is ready. File handlers are called after X event handlers.

Timer events

Register a callback to occur at some time in the future with Tk_CreateTimerHandler. The handler is only called once. If you need to delete the handler before it gets called, use Tk_DeleteTimerHandler. These are described in the Tk_TimerToken man page.

Idle callbacks

If there are no outstanding events, the Tk makes idle callbacks before waiting for new events to arrive. In general, Tk widgets queue their display routines to be called at idle time. Use Tk_DoWhenIdle to queue an idle callback, and use Tk_CancelIdleCall to remove the callback from the queue. These are described in the DoWhenIdle man page.

Sleeping

The Tk_Sleep procedure delays execution for a specified number of milliseconds. It is described in the Sleep man page.

Reporting script errors

If you widget makes a callback into the script level, what do you do when the callback returns an error? Use the Tk_BackgroundError procedure that invokes the standard tkerror procedure to report the error to the user. This is described in the BackgdErr man page.

Handling X protocol errors

You can handle X protocol errors by registering a handler with Tk_CreateErrorHandler. Unregister it with Tk_DeleteErrorHandler. These are described in the CrtErrHdlr man page. Because X has an asynchronous interface, the error will be reported sometime after the offending call was made. You can call the Xlib XSynchronize routine to turn off the asynchronous behavior in order to help you debug.

Using the X resource database.

The Tk_GetOption procedure looks up items in the X resource database. This is described in the GetOption man page.

The resource class of a window is set with Tk_SetClass, and the current class setting is retrieved with Tk_Class. These are described in the SetClass man page.

Managing bitmaps

Tk maintains a registry of bitmaps by name, e.g. `gray50` and `questhead`. You can define new bitmaps with `Tk_DefineBitmap`, and you can get a handle on the bitmap from its name with `Tk_GetBitmap`. Related procedures include `Tk_NameOfBitmap`, `Tk_SizeOfBitmap`, `Tk_FreeBitmap`, and `Tk_GetBitmapFromData`. These are described in the `GetBitmap` man page.

Creating new image types

The `Tk_CreateImageType` procedure is used to register the implementation of a new image type. The registration includes several procedures that callback into the implementation to support creation, display, and deletion of images. The interface to an image implementation is described in the `CrtImgType` man page.

When an image changes, the widgets that display it are notified by calling `Tk_ImgChanged`. This is described in the `ImgChanged` man page.

Using an image in a widget

The following routines support widgets that display images. `Tk_GetImage` maps from the name to a `Tk_Image` data structure. `Tk_RedrawImage` causes the image to update its display. `Tk_SizeOfImage` tells you how big it is. When the image is no longer in use, call `Tk_FreeImage`. These are described in the `GetImage` man page.

Photo image types

One of the image types is `photo`, which has its own C interface for defining new formats. The job of a format handler is to read and write different image formats such as `GIF` or `JPEG` so that the `photo` image can display them. The `Tk_CreatePhotoImageFormat` procedure sets up the interface, and it is described in the `CrtPhImgFmt` man page.

There are several support routines for photo format handlers. The `Tk_FindPhoto` procedure maps from a photo name to its associated `Tk_PhotoHandle` data structure. The image is updated with `Tk_PhotoBlank`, `Tk_PhotoPutBlock`, and `Tk_PhotoPutZoomedBlock`. The image values can be obtained with `Tk_PhotoGetImage`. The size of the image can be manipulated with `Tk_PhotoExpand`, `Tk_PhotoGetSize`, and `Tk_PhotoSetSize`. These support routines are described in the `FindPhoto` man page.

Canvas object support

The C interface for defining new canvas items is exported via the `Tk_CreateItemType` procedure. The description for a canvas item includes a set of procedures that the canvas widget uses to call the implementation of the canvas item type. This interface is described in detail in the `CrtItemType` man page.

There are support routines for the managers of new item types. The CanvTkwin man page describes Tk_CanvasTkwin, Tk_CanvasGetCoord, Tk_CanvasDrawableCoords, Tk_CanvasSetStippleOrigin, Tk_CanvasWindowCoords, and Tk_CanvasEventuallyRedraw. The following procedures help with the generation of postscript: Tk_CanvasPsY, Tk_CanvasPsBitmap, Tk_CanvasPsColor, Tk_CanvasPsFont, Tk_CanvasPsPath, and Tk_CanvasPsStipple. These are described by the CanvPsY man page. If you are manipulating text items directly, then you can use the Tk_CanvasTextInfo procedure to get a description of the selection state and other details about the text item. This procedure is described in the CanvTxtInfo man page.

Geometry managment

A widget requests a certain size with the Tk_GeometryRequest procedure. If it draws a border inside that area, it calls Tk_SetInternalBorder. The geometery manager responds to these requests, although the widget may get a different size. These are described in the GeomReq man page.

The Tk_ManageGeometry procedure sets up the relationship between the geometry manager and a widget. This is described in the ManageGeom man page.

The Tk_MaintainGeometry arranges for one window to stay at a fixed position relative to another widget. This is used by the place geometry manager. The relationship is broken with the Tk_UnmaintainGeometry call. These are described in the MaintGeom man page.

The Tk_SetGrid enabled gridded geometry management. The grid is turned off with Tk_UnsetGrid. These are described in the SetGrid man page.

String identifiers (UIDS)

Tk maintains a database of string values such that a string only appears in it once. The Tk_Uid type refers to such a string. You can test for equality by using the value of Tk_Uid, which is the strings address, as an identifier. A Tk_Uid is used as a name in the various GetByName calls introduced below. The Tk_GetUid procedure installs a string into the registry. It is described in the GetUid man page.

Colors and Colormaps

Use Tk_GetColor and Tk_GetColorByValue to allocate a color. You can retrieve the string name of a color with Tk_NameOfColor. When you are done using a color you need to call Tk_FreeColor. Colors are shared among widgets, so it is important to free them when you are done using them. These are described in the GetColor man page.

Use Tk_GetColormap and Tk_FreeColormap to allocate and free a colormap. Colormaps are shared, if possible, so you should use these routines instead of the lower -level X routines to allocate colormaps. These are described in the GetClrmap man page.

The color model used by the screen can be set and queried with Tk_SetColorModel and Tk_GetColorModel. For example, you can force a window into monochrome mode when it runs on a color screen. These are described in the SetCModel man page.

The window's visual type is set with Tk_SetWindowVisual. This is described in the SetVisual man page.

3D Borders

The three dimensional relief used for widget borders is supported by a collection of routines described by the 3DBorder man page. The routines are Tk_Get3DBorder, Tk_Draw3DRectangle, Tk_Fill3DRectangle, Tk_Draw3DPolygon, Tk_Fill3DPolygon, Tk_3DVerticalBevel, Tk_3DHorizontalBevel, Tk_SetBackgroundFromBorder, Tk_NameOf3DBorder, Tk_3DBorderColor, Tk_3DBorderGC, and Tk_Free3DBorder.

Mouse cursors

Allocate a cursor with Tk_GetCursor and Tk_GetCursorFromData. Map back to the name of the cursor with Tk_NameOfCursor. Release the cursor resource with Tk_FreeCursor. These are described in the GetCursor man page.

Font structures

Allocate a font with Tk_GetFontStruct. Get the name of a font with Tk_NameOfFontStruct. Release the font with Tk_FreeFontStruct. These are described in the GetFontStr man page.

Graphics Contexts

Allocate a graphics context with Tk_GetGC, and free it with Tk_FreeGC. These are described in the GetGC man page.

Allocate a pixmap

Allocate and free pixmaps with Tk_GetPixmap and Tk_FreePixmap. These are described in the GetPixmap man page.

Screen measurements

Translate between strings like 4c or 72p and screen distances with Tk_GetPixels and Tk_GetScreenMM. The first call returns pixels (integers), the second returns millimeters as a floating point number. These are described in the GetPixels man page.

Relief style

Translate between relief styles and names with Tk_GetRelief and Tk_NameOfRelief. These are described in the GetRelief man page.

Text anchor positions

Translate between strings and anchor positions with Tk_GetAnchor and Tk_NameOfAnchor. These are described in the GetAnchor man page.

Line cap styles

Translate between line cap styles and names with Tk_GetCapStyle and Tk_NameOfCapStyle. These are described in the GetCapStyl man page.

Line join styles

Translate between line join styles and names with Tk_GetJoinStyle and Tk_NameOfJoinStyle. These are described in the GetJoinStl man page.

Text justification styles

Translate between line justification styles and names with Tk_GetJustify and Tk_NameOfJustify. These are described in teh GetJustify man page.

Atoms

An atom is an integer that references a string that has been registered with the X server . Tk maintians a cache of the atom registry to avoid contacting the X server when atoms are used. Use Tk_InternAtom to install an atom in the registry , and Tk_GetAtomName to return the name given an atom. These are described by the InternAtom man page.

X resource ID management

Each X resource like a color or pixmap has a resource ID associated with it. The Tk_FreeXId call releases an ID so it can be reused. This is used, for example, by routines like Tk_FreeColor and Tk_FreePixmap. It is described in the FreeXId man page.

CHAPTER 31

Writing a Tk Widget in C

This chapter describes in the implementation of a simple clock widget.

A custom widget implemented in C has the advantage of being effecient and flexible. However, it is more work, too. This chapter illustrates the effort by explaining the implementation of a clock widget.

Implementing a New Widget

This section describes the implementation of a clock Tk widget. This is just a digital clock that displays the current time according to a format string. The formatting is done by the `strftime` library, so you can use any format supported by that routine. The default format is `%H:%M:%S`, which results in `16:23:45`.

The implementation of a widget includes several parts.

- A data structure to describe one instance of the widget.
- A set of configuration options for the widget.
- A command procedure to create a new instance of the widget.
- A command procedure ot operate on an instance of the widget.
- A configuration procedure used when creating and reconfiguring the widget.
- An event handling procedure.
- A display procedure.
- Other widget-specific procedures.

The Widget Data Structure

Each widget is associated with a data structure that describes it. Any widget structure will need a pointer to the Tcl interpreter, the Tk window, and the X display. The interpreter is used in many of the Tcl and Tk library calls, and it provides a way to call out to the script or query and set Tcl variables. The Tk window is needed for various Tk operations, and the X display is used when doing low-level graphic operations. The rest of the information in the data structure depends on the widget. The structure for the clock widget is given below. The different types will be explained as they are used in the rest of the code.

Example 31–1 The `Clock` widget data structure.

```
#include "tkPort.h"
#include "tk.h"

typedef struct {
    Tk_Window tkwin;        /* The window for the widget */
    Display *display;       /* X's handle on the display */
    Tcl_Interp *interp;     /* Interpreter of the widget */
    /*
     * Clock-specific attributes.
     */
    int borderWidth;        /* Size of 3-D border */
    int relief;             /* Style of 3-D border */
    Tk_3DBorder background;/* Color for border, background */
    XColor *foreground;     /* Color for the text */
    XColor *highlight;      /* Color for the highlight */
    int highlightWidth;     /* Thickness of highlight rim */
    XFontStruct *fontPtr;   /* Font info for the text */
    char *format;           /* Format for the clock text */
    /*
     * Graphic contexts and other support.
     */
    GC highlightGC;         /* Highlight graphics context */
    GC textGC;              /* Text graphics context */
    Tk_TimerToken token;    /* For periodic callbacks */
    char *clock;            /* Pointer to the clock string */
    int numChars;           /* in the text */
    int textWidth;          /* in pixels */
    int textHeight;         /* in pixels */
    int flags;              /* Flags defined below */
} Clock;
/*
 * Flag bit definitions.
 */
#define REDRAW_PENDING      0x1
#define GOT_FOCUS           0x2
#define TICKING  0x4
```

Specifying Widget Attributes

Several of the fields in the Clock structure are attributes that can be set when the widget is create or reconfigured with the configure operation. The default values, their resource names, and their class names are specified with an array of `Tk_ConfigSpec` records, and this array is processed by the `Tk_ConfigureWidget` operation. The specifications for the Clock structure are given in the next example.

Example 31–2 Configuration specs for the clock widget.

```
static Tk_ConfigSpec configSpecs[] = {
   {TK_CONFIG_BORDER, "-background", "background",
      "Background","light blue",
      Tk_Offset(Clock, background), TK_CONFIG_COLOR_ONLY},
   {TK_CONFIG_BORDER, "-background", "background",
      "Background","white", Tk_Offset(Clock, background),
      TK_CONFIG_MONO_ONLY},
   {TK_CONFIG_SYNONYM, "-bg", "background", (char *) NULL,
      (char *) NULL, 0, 0},

   {TK_CONFIG_SYNONYM, "-bd", "borderWidth", (char *) NULL,
      (char *) NULL, 0, 0},
   {TK_CONFIG_PIXELS, "-borderwidth", "borderWidth",
      "BorderWidth","2", Tk_Offset(Clock, borderWidth), 0},
   {TK_CONFIG_RELIEF, "-relief", "relief", "Relief",
      "ridge", Tk_Offset(Clock, relief), 0},

   {TK_CONFIG_COLOR, "-foreground", "foreground",
      "Foreground","black", Tk_Offset(Clock, foreground),
      0},
   {TK_CONFIG_SYNONYM, "-fg", "foreground", (char *) NULL,
      (char *) NULL, 0, 0},

   {TK_CONFIG_COLOR, "-highlightcolor", "highlightColor",
      "HighlightColor","red", Tk_Offset(Clock, highlight),
      TK_CONFIG_COLOR_ONLY},
   {TK_CONFIG_COLOR, "-highlightcolor", "highlightColor",
       "HighlightColor","black",
      Tk_Offset(Clock, highlight),TK_CONFIG_MONO_ONLY},
   {TK_CONFIG_PIXELS, "-highlightthickness",
      "highlightThickness","HighlightThickness",
      "2", Tk_Offset(Clock, highlightWidth), 0},

   {TK_CONFIG_STRING, "-format", "format", "Format",
      "%H:%M:%S", Tk_Offset(Clock, format), 0},
   {TK_CONFIG_FONT, "-font", "font", "Font",
      "*courier-medium-r-normal-*-18-*",
       Tk_Offset(Clock, fontPtr), 0},

   {TK_CONFIG_END, (char *) NULL, (char *) NULL,
       (char *) NULL, (char *) NULL, 0, 0}
```

```
};
```

The initial field is a type, such as TK_CONFIG_BORDER. Colors and borders will be explained shortly. The next field is the command line flag for the attribute, e.g. -background. Then comes the resource name and the class name. The default value is next, e.g., light blue. The offset of a structure member is next, and the Tk_Offset macro is used to compute this offset. The last field is a bitmask fo flags. The two used in this example are TK_CONFIG_COLOR_ONLY and TK_CONFIG_MONO_ONLY, which restrict the application of the configuration setting to color and monochrome displays, respectively. You can define additional flags and pass them into Tk_ConfigureWidget if you have a family of widgets that share most, but not all, of their attributes. The tkButton.c file in the Tk sources has an example of this.

Table 31–1 lists the correspondence bewtween the configuration type passed Tk_ConfigureWidget and the type of the associated field in the widget data structure. The complete details are given in the ConfigWidg man page. Some of the table entries reference a Tk procedure like Tk_GetCapStyle. In those cases an integer-valued field takes on a few limited values that are described in the man page for that procedure.

Table 31–1 Configuration flags and corresponding C types.

TK_CONFIG_ACTIVE_CURSOR	Cursor
TK_CONFIG_ANCHOR	Tk_Anchor
TK_CONFIG_BITMAP	Pixmap
TK_CONFIG_BOOLEAN	int (0 or 1)
TK_CONFIG_BORDER	Tk_3DBorder *
TK_CONFIG_CAP_STYLE	int (see Tk_GetCapStyle)
TK_CONFIG_COLOR	XColor *
TK_CONFIG_CURSOR	Cursor
TK_CONFIG_CUSTOM	
TK_CONFIG_DOUBLE	double
TK_CONFIG_END	(signals end of options)
TK_CONFIG_FONT	XFontStruct *
TK_CONFIG_INT	int
TK_CONFIG_JOIN_STYLE	int (see Tk_GetJoinStyle)
TK_CONFIG_JUSTIFY	Tk_Justify
TK_CONFIG_MM	double
TK_CONFIG_PIXELS	int

Table 31–1 Configuration flags and corresponding C types.

TK_CONFIG_RELIEF	int (see Tk_GetRelief)
TK_CONFIG_STRING	char *
TK_CONFIG_SYNONYM	(alias for other option)
TK_CONFIG_UID	Tk_Uid
TK_CONFIG_WINDOW	Tk_Window

The Widget Class Command

The Tcl command that creates an instance of a widget is known as the class command. In our example, the `clock` command creates a clock widget. The command procedure for the `clock` command is shown below. The procedure allocates the `Clock` data structure. It registers an event handler that gets called when the widget is exposed, resized, or gets focus. It creates a new Tcl command that operates on the wiget. Finally, it calls `ClockConfigure` to set up the widget according to the attributes specified on the command line and the default configuration specifications.

Example 31–3 The `ClockCmd` command procedure.

```
int
ClockCmd(clientData, interp, argc, argv)
    ClientData clientData;  /* Main window of the app */
    Tcl_Interp *interp;     /* Current interpreter. */
    int argc;               /* Number of arguments. */
    char **argv;            /* Argument strings. */
{
    Tk_Window main = (Tk_Window) clientData;
    Clock *clockPtr;
    Tk_Window tkwin;

    if (argc < 2) {
        Tcl_AppendResult(interp, "wrong # args: should be '",
             argv[0], " pathName ?options?'", (char *) NULL);
        return TCL_ERROR;
    }
    tkwin = Tk_CreateWindowFromPath(interp, main,
             argv[1], (char *) NULL);
    if (tkwin == NULL) {
        return TCL_ERROR;
    }
    Tk_SetClass(tkwin, "Clock");
    /*
     * Allocate and initialize the widget record.
     */
    clockPtr = (Clock *) ckalloc(sizeof(Clock));
    clockPtr->tkwin = tkwin;
    clockPtr->display = Tk_Display(tkwin);
```

```
    clockPtr->interp = interp;
    clockPtr->borderWidth = 0;
    clockPtr->highlightWidth = 0;
    clockPtr->relief = TK_RELIEF_FLAT;
    clockPtr->background = NULL;
    clockPtr->foreground = NULL;
    clockPtr->highlight = NULL;
    clockPtr->fontPtr = NULL;
    clockPtr->textGC = None;
    clockPtr->highlightGC = None;
    clockPtr->token = NULL;
    clockPtr->clock = NULL;
    clockPtr->numChars = 0;
    clockPtr->textWidth = 0;
    clockPtr->textHeight = 0;
    clockPtr->flags = 0;
    /*
     * Register a handler for when the window is
     * exposed or resized.
     */
    Tk_CreateEventHandler(clockPtr->tkwin,
       ExposureMask|StructureNotifyMask|FocusChangeMask,
       ClockEventProc, (ClientData) clockPtr);
    /*
     * Create a Tcl command that operates on the widget.
     */
    Tcl_CreateCommand(interp, Tk_PathName(clockPtr->tkwin),
       ClockInstanceCmd,
       (ClientData) clockPtr, (void (*)()) NULL);
    /*
     * Parse the command line arguments.
     */
    if (ClockConfigure(interp, clockPtr,
            argc-2, argv+2, 0) != TCL_OK) {
       Tk_DestroyWindow(clockPtr->tkwin);
       return TCL_ERROR;
    }
    interp->result = Tk_PathName(clockPtr->tkwin);
    return TCL_OK;
}
```

Widget Instance Command

For each instance of a widget a new command is created that operates on that widget. This is called the widget instance command. Its name is the same as the Tk pathname of the widget. In the clock example, all that is done on instances is to query and change their attributes. Most of the work is done by `Tk_ConfigureWidget` and `ClockConfigure`, which is shown in the next section. The `ClockInstanceCmd` command procedure is shown in the next exmaple.

Example 31–4 The `ClockInstanceCmd` command procedure.

```
static int
ClockInstanceCmd(clientData, interp, argc, argv)
    ClientData clientData;  /* A pointer to a Clock struct */
    Tcl_Interp *interp;     /* The interpreter */
    int argc;               /* The number of arguments */
    char *argv[];           /* The command line arguments */
{
    Clock *clockPtr = (Clock *)clientData;
    int result = TCL_OK;
    char c;
    int length;

    if (argc < 2) {
        Tcl_AppendResult(interp, "wrong # args: should be '",
            argv[0], " option ?arg arg ...?'", (char *) NULL);
        return TCL_ERROR;
    }
    c = argv[1][0];
    length = strlen(argv[1]);
    if ((c == 'c') && (strncmp(argv[1], "cget", length) == 0)
            && (length >= 2)) {
        if (argc != 3) {
            Tcl_AppendResult(interp,
                 "wrong # args: should be '",
                argv[0], " cget option'",
                (char *) NULL);
            return TCL_ERROR;
        }
        result = Tk_ConfigureValue(interp, clockPtr->tkwin,
             configSpecs, (char *) clockPtr, argv[2], 0);
    } else if ((c == 'c') &&
            (strncmp(argv[1], "configure", length) == 0)
            && (length >= 2)) {
        if (argc == 2) {
            /*
             * Return all configuration information.
             */
            result = Tk_ConfigureInfo(interp, clockPtr->tkwin,
                configSpecs, (char *) clockPtr,
                (char *) NULL,0);
        } else if (argc == 3) {
            /*
             * Return info about one attribute, like cget.
             */
            result = Tk_ConfigureInfo(interp, clockPtr->tkwin,
                configSpecs, (char *) clockPtr, argv[2], 0);
        } else {
            /*
             * Change one or more attributes.
             */
            result = ClockConfigure(interp, clockPtr, argc-2,
                argv+2,TK_CONFIG_ARGV_ONLY);
        }
```

```
    } else {
        Tcl_AppendResult(interp, "bad option '", argv[1],
            "': must be cget, configure, position, or size",
            (char *) NULL);
        return TCL_ERROR;
    }
    return result;
}
```

Configuring And Reconfiguring Attributes

When the widget is created or reconfigured, then the implementation needs to allocate the resources implied by the attribute settings. Each clock widget uses some colors and a font. These are described by graphics contexts. A graphic context is used by X to parameterize graphic operations. Instead of specifying every possible attribute in the X calls, a graphics context is initialized with a subset of the parameters and this is passed into the X drawing commands. The context can specify the foreground and background colors, clip masks, line styles, and so on. In the example, two different graphics contexts are used, one for the highlight rectangle and one for the text and background. They use different colors, so different contexts are needed. The graphics contexts are allocated once and reused each time the widget is displayed.

There are two kinds of color resources used by the widget. The focus highlight and the text foreground are simple colors. The background is a Tk_3DBorder, which is a set of colors used to render 3D borders. The background color is specified in the attribute, and the other colors are computed based on that color. The code uses `Tk_3DBorderColor` to map back to the originial color for use in the background of the widget.

After the resources are set up, a call to redisplay the widget is scheduled for the next idle period. This is a standard idiom for Tk widgets. It means that you can create and reconfigure a widget in the middle of a script, and all the changes only result in one redisplay. The `REDRAW_PENDING` flag is used to ensure that only one redisplay is queued up at any time. The `ClockConfigure` procedure is shown in the next example.

Example 31–5 `ClockConfigure` allocates resources for the widget.

```
static int
ClockConfigure(interp, clockPtr, argc, argv, flags)
    Tcl_Interp *interp;/* Needed for return values and errors
*/
    Clock *clockPtr; /* The per-instance data structure */
    int argc;        /* Number of valid entries in argv */
    char *argv[];    /* The command line arguments */
    int flags;       /* Tk_ConfigureClock flags */
{
    XGCValues gcValues;
```

```
GC newGC;

/*
 * Tk_ConfigureWidget parses the command line arguments
 * and looks for defaults in the resource database.
 */
if (Tk_ConfigureWidget(interp, clockPtr->tkwin,
       configSpecs, argc, argv, (char *) clockPtr, flags)
           != TCL_OK) {
   return TCL_ERROR;
}
/*
 * Give the widget a default background so it doesn't get
 * a random background between the time it is initially
 * displayed by the X server and we paint it
 */
Tk_SetWindowBackground(clockPtr->tkwin,
   Tk_3DBorderColor(clockPtr->background)->pixel);
/*
 * Set up the graphics contexts to display the widget.
 * These contexts are all used to draw off-screen
 * pixmaps, so turn off exposure notifications.
 */
gcValues.graphics_exposures = False;
gcValues.background = clockPtr->highlight->pixel;
newGC = Tk_GetGC(clockPtr->tkwin,
   GCBackground|GCGraphicsExposures, &gcValues);
if (clockPtr->highlightGC != None) {
   Tk_FreeGC(clockPtr->display, clockPtr->highlightGC);
}
clockPtr->highlightGC = newGC;

gcValues.background =
   Tk_3DBorderColor(clockPtr->background)->pixel;
gcValues.foreground = clockPtr->foreground->pixel;
gcValues.font = clockPtr->fontPtr->fid;
newGC = Tk_GetGC(clockPtr->tkwin,
   GCBackground|GCForeground|GCFont|GCGraphicsExposures,
   &gcValues);
if (clockPtr->textGC != None) {
   Tk_FreeGC(clockPtr->display, clockPtr->textGC);
}
clockPtr->textGC = newGC;

/*
 * Determine how big the widget wants to be.
 */
ComputeGeometry(clockPtr);

/*
 * Set up a call to display ourself.
 */
if ((clockPtr->tkwin != NULL) &&
        Tk_IsMapped(clockPtr->tkwin)
       && !(clockPtr->flags & REDRAW_PENDING)) {
```

```
        Tk_DoWhenIdle(ClockDisplay, (ClientData) clockPtr);
        clockPtr->flags |= REDRAW_PENDING;
    }
    return TCL_OK;
}
```

Displaying The Clock

There are two parts to a widget's display. First the size must be determined. This is done at configuration time, and then that space is requested from the geometry manager. When the widget is later displayed, it should use the `Tk_Width` and `Tk_Height` calls to find out how much space it was actually allocated by the geometry manager. The next example shows `ComputeGeometry`.

Example 31–6 `ComputeGeometry` figures out how big the widget is.

```
static void
ComputeGeometry(Clock *clockPtr)
{
    int width, height;
    struct tm *tmPtr;       /* Time info split into fields */
    struct timeval tv;      /* BSD-style time value */
    int offset = clockPtr->highlightWidth +
                 clockPtr->borderWidth
                 + 2;       /* Should be padX attribute */
    char clock[1000];

    /*
     * Get the time and format it to see how big it will be.
     * gettimeofday returns the current time.
     * localtime parses this into day, hour, etc.
     * strftime formats this into a string according to
     * a format. By default we use %H:%M:%S
     */
    gettimeofday(&tv, NULL);
    tmPtr = localtime(&tv.tv_sec);
    strftime(clock, 1000, clockPtr->format, tmPtr);
    if (clockPtr->clock != NULL) {
        ckfree(clockPtr->clock);
    }
    clockPtr->clock = ckalloc(1+strlen(clock));
    clockPtr->numChars = strlen(clock);
    /*
     * Let Tk tell us how big the string will be.
     */
    TkComputeTextGeometry(clockPtr->fontPtr, clock,
        clockPtr->numChars, 0, &clockPtr->textWidth,
        &clockPtr->textHeight);
    width = clockPtr->textWidth + 2*offset;
    height = clockPtr->textHeight + 2*offset;
    /*
```

```
     * Request size and border from the geometry manager.
     */
    Tk_GeometryRequest(clockPtr->tkwin, width, height);
    Tk_SetInternalBorder(clockPtr->tkwin, offset);
}
```

Finally we get to the actual display of the widget! The routine is careful to check that the widget still exists and is mapped. This is important because the redisplay is scheduled asynchronously . The current time is converted to a string. This uses library procedures that exist on SunOS. There might be different routines on your system. The string is painted into a pixmap, which is a drawable region of memory that is off-screen. After the whole display has been painted, the pixmap is copied into on-screen memory to avoid flickering as the image is cleared and repainted. The text is painted first, then the borders. This ensures that the borders overwrite the text if the widget has not been allocated enough room by the geometry manager .

This example allocates and frees the off-screen pixmap for each redisplay . This is the standard idiom for Tk widgets. They temporarily allocate the off-screen pixmap each time they redisplay . In the case of a clock that updates every second, it might be reasonable to permanently allocate the pixmap and store its pointer in the Clock data structure. Make sure to reallocate the pixmap if the size changes.

After the display is finished, another call to the display routine is scheduled to happen in one second. If you were to embellish this widget, you might want to make the uptime period a paramter . The TICKING flag is used to note that the timer callback is scheduled. It is checked when the widget is destroyed so that the callback can be canceled. The next example shows `ClockDisplay`.

Example 31–7 The `ClockDisplay` procedure.

```
static void
ClockDisplay(ClientData clientData)
{
    Clock *clockPtr = (Clock *)clientData;
    Tk_Window tkwin = clockPtr->tkwin;
    Pixmap pixmap;
    int offset, x, y;
    struct tm *tmPtr;/* Time info split into fields */
    struct timeval tv;/* BSD-style time value */
    /*
     * Make sure the button still exists
     * and is mapped onto the display before painting.
     */
    clockPtr->flags &= ~(REDRAW_PENDING|TICKING);
    if ((clockPtr->tkwin == NULL) || !Tk_IsMapped(tkwin)) {
        return;
    }
    /*
     * Format the time into a string.
     * localtime chops up the time into fields.
```

```
 * strftime formats the fields into a string.
 */
gettimeofday(&tv, NULL);
tmPtr = localtime(&tv.tv_sec);
strftime(clockPtr->clock, clockPtr->numChars+1,
   clockPtr->format, tmPtr);
/*
 * To avoid flicker when the display is updated, the new
 * image is painted in an offscreen pixmap and then
 * copied onto the display in one operation.
 */
pixmap = Tk_GetPixmap(clockPtr->display,
   Tk_WindowId(tkwin),Tk_Width(tkwin),
   Tk_Height(tkwin), Tk_Depth(tkwin));
Tk_Fill3DRectangle(clockPtr->display, pixmap,
   clockPtr->background, 0, 0, Tk_Width(tkwin),
   Tk_Height(tkwin), 0, TK_RELIEF_FLAT);
/*
 * Paint the text first.
 */
offset = clockPtr->highlightWidth +
         clockPtr->borderWidth;
x = (Tk_Width(tkwin) - clockPtr->textWidth)/2;
if (x < 0) x = 0;
y = (Tk_Height(tkwin) - clockPtr->textHeight)/2;
if (y < 0) y = 0;

TkDisplayText(clockPtr->display, pixmap,
   clockPtr->fontPtr, clockPtr->clock,
   clockPtr->numChars, x, y, clockPtr->textWidth,
   TK_JUSTIFY_CENTER, -1, clockPtr->textGC);
/*
 * Display the borders, so they overwrite any of the
 * text that extends to the edge of the display.
 */
if (clockPtr->relief != TK_RELIEF_FLAT) {
   Tk_Draw3DRectangle(clockPtr->display, pixmap,
      clockPtr->background, clockPtr->highlightWidth,
      clockPtr->highlightWidth,
      Tk_Width(tkwin) - 2*clockPtr->highlightWidth,
      Tk_Height(tkwin) - 2*clockPtr->highlightWidth,
      clockPtr->borderWidth, clockPtr->relief);
}
if (clockPtr->highlightWidth != 0) {
   GC gc;
   if (clockPtr->flags & GOT_FOCUS) {
      gc = clockPtr->highlightGC;
   } else {
      gc = Tk_3DBorderGC(clockPtr->background,
         TK_3D_FLAT_GC);
   }
   TkDrawFocusHighlight(tkwin, gc,
      clockPtr->highlightWidth, pixmap);
}
/*
```

```
     * Copy the information from the off-screen pixmap onto
     * the screen, then delete the pixmap.
     */
    XCopyArea(clockPtr->display, pixmap, Tk_WindowId(tkwin),
        clockPtr->textGC, 0, 0, Tk_Width(tkwin),
        Tk_Height(tkwin), 0, 0);
    Tk_FreePixmap(clockPtr->display, pixmap);
    /*
     * Queue another call to ourselves.
     */
    clockPtr->token = Tk_CreateTimerHandler(1000,
        ClockDisplay, (ClientData)clockPtr);
    clockPtr->flags |= TICKING;
}
```

The Window Event Procedure

Each widget registers an event handler for expose and resize events. If it implements and focus highlight, it also needs to be notified of focus events. If you have used other toolkits, you may expect to register callbacks for mouse and keystroke events too. You shouldn't have to do that. Instead, use the regular Tk bind facility and define your bindings in Tcl. That way they can be customized by applications.

Example 31–8 The `ClockEventProc` handles window events.

```
static void
ClockEventProc(ClientData clientData, XEvent *eventPtr)
{
    Clock *clockPtr = (Clock *) clientData;
    if ((eventPtr->type == Expose) &&
        (eventPtr->xexpose.count == 0)) {
            goto redraw;
    } else if (eventPtr->type == DestroyNotify) {
        Tcl_DeleteCommand(clockPtr->interp,
            Tk_PathName(clockPtr->tkwin));
        /*
         * Zapping the tkwin lets the other procedures
         * know we are being destroyed.
         */
        clockPtr->tkwin = NULL;
        if (clockPtr->flags & REDRAW_PENDING) {
            Tk_CancelIdleCall(ClockDisplay,
                (ClientData) clockPtr);
            clockPtr->flags &= ~REDRAW_PENDING;
        }
        if (clockPtr->flags & TICKING) {
            Tk_DeleteTimerHandler(clockPtr->token);
            clockPtr->flags &= ~TICKING;
        }
        /*
```

```
         * This results in a call to ClockDestroy.
         */
        Tk_EventuallyFree((ClientData) clockPtr,
            ClockDestroy);
    } else if (eventPtr->type == FocusIn) {
        if (eventPtr->xfocus.detail != NotifyPointer) {
            clockPtr->flags |= GOT_FOCUS;
            if (clockPtr->highlightWidth > 0) {
                goto redraw;
            }
        }
    } else if (eventPtr->type == FocusOut) {
        if (eventPtr->xfocus.detail != NotifyPointer) {
            clockPtr->flags &= ~GOT_FOCUS;
            if (clockPtr->highlightWidth > 0) {
                goto redraw;
            }
        }
    }
    return;
redraw:
    if ((clockPtr->tkwin != NULL) &&
            !(clockPtr->flags & REDRAW_PENDING)) {
        Tk_DoWhenIdle(ClockDisplay, (ClientData) clockPtr);
        clockPtr->flags |= REDRAW_PENDING;
    }
}
```

Final Cleanup

When a widget is destroyed you need to free up any resources it has allocated. The resources associated with attributes are cleaned up by `Tk_FreeOptions`. The others you must take care of yourself. The `ClockDestroy` procedure is called as a result fo the `Tk_EventuallyFree` call in the `ClockEventProc`. The `Tk_EventuallyFree` procedure is part of a protocol that is needed for widgets that might get deleted when in the middle of processing. Typically the `Tk_Preserve` and `Tk_Release` procedures are called at the beginning and end of the widget instance command to mark the widget as being in use. `Tk_EventuallyFree` will wait until `Tk_Release` is called before calling the cleanup procedure. The next example shows `ClockDestroy`.

Example 31–9 The `ClockDestroy` cleanup procedure.

```
static void
ClockDestroy(clientData)
 ClientData clientData;/* Info about entry widget. */
{
    register Clock *clockPtr = (Clock *) clientData;

    /*
```

```
     * Free up all the stuff that requires special handling,
     * then let Tk_FreeOptions handle resources associated
     * with the widget attributes.
     */
    if (clockPtr->highlightGC != None) {
       Tk_FreeGC(clockPtr->display, clockPtr->highlightGC);
    }
    if (clockPtr->textGC != None) {
       Tk_FreeGC(clockPtr->display, clockPtr->textGC);
    }
    if (clockPtr->clock != NULL) {
       ckfree(clockPtr->clock);
    }
    if (clockPtr->flags & TICKING) {
       Tk_DeleteTimerHandler(clockPtr->token);
    }
    if (clockPtr->flags & REDRAW_PENDING) {
       Tk_CancelIdleCall(ClockDisplay,
           (ClientData) clockPtr);
    }
    /*
     * This frees up colors and fonts and any allocated
     * storage associated with the widget attributes.
     */
    Tk_FreeOptions(configSpecs, (char *) clockPtr,
       clockPtr->display, 0);
    ckfree((char *) clockPtr);
}
```

CHAPTER 32

Tcl Extension Packages

This chapter surveys a few of the more popular Tcl extension packages.

Extension packages add suites of Tcl commands, usually as a combination of new built-in commands written in C and associated Tcl procedures. Some extensions provide new Tk widgets and geometry managers. This chapter surveys a few of the more popular extensions. Some are complex enough to deserve their own book, so this chapter is just meant to give you a feel for what these packages have to offer. For the details, you will have to consult the documentation that comes with the packages. This chapter briefly describes the following packages.

- *Extended Tcl* adds commands that provide access to more Unix libraries and system calls. It adds new list operations and new loop constructs. It adds profiling commands so you can analyze the performance of your Tcl scripts.
- *Expect* adds commands that let you control interactive programs. Programs that insist on having a conversation with a user can be fooled by *expect* into doing work for you automatically.
- *Tcl debugger*. Part of the Expect package includes a small Tcl debugger that lets you set breakpoints and step through scripts.
- *Tcl-dp* adds commands that set up network connections among Tcl interpreters. You can set up distributed systems using *Tcl-dp*.
- *BLT* provides a table geometry manager for Tk, a graph widget, and more.
- [incr tcl] provides an object system for Tcl. The scope for variables and procedures can be limited by using classes, and multiple inheritence can be

used to set up a class hierarchy. The Tk-like interface with attributes and values is well supported by the package so you can create mega-widgets that look and feel like native Tk widgets to the programmer.

There are many more extentions available on the internet, and there is not enough time or space to describe even these extentions in much detail. This chapter provides a few tips on how to integrate these extensions into your application and what they can provide for you.

Extended Tcl

Extended Tcl, or tclX, provides many new built-in commands and support procedures. It provides access to more UNIX system calls and librarys, and it provides tools that are useful for developing large Tcl applications. Over time, features from tclX have been adopted by Ousterhout for use in the core language. For example, arrays and the `addinput` command originated from tclX.

The tclX extension is a little different from other applications because it assumes a more fundamental role. It provides its own script library mechanism, which is described in more detail below, and its own interactive shell. The extneded Tcl shell is normally installed as *tcl*, and the Extended Tcl/Tk shell is normally installed as *wishx*.

There is one main manual page for tclX that describes all the commands and Tcl procedures provided by the package. The system also comes with a built-in help system so you can easily browse the man pages for standard Tcl and Extended tcl. The *tclhelp* program provides a graphical interface to the help system, or use the `help` command when running under the Extended Tcl shell, *tcl*.

Extended Tcl was designed and implemented by Karl Lehenbauer and Mark Diekhans, with help in the early stages from Peter da Silva. Extended Tcl is freely redistributable, including for commercial use and resale. You can fetch the tclX distribution from the following FTP site:

```
ftp.neosoft.com:/pub/tcl/distrib/tclX7.4a.tar.gz
```

Adding tclX to your application

TclX has a different script library mechanism that makes integrating it into your application a little different that other extension packages. The main thing is that you need to call `TclX_Init` in your `Tcl_AppInit` procedure, not the standard `Tcl_Init` procedure. A version of the `tclAppInit.c` that is oriented towards Extended Tcl is provided with its distribution. The tclX library facility can read the `tclIndex` files of the standard library mechanism, so you can still use other packages.

It is possible, but rather awkward, to use the tclX commands and procedures with the standard library mechanism, which is described in Chapter 9. Instead of calling `TclX_Init`, you call `TclXCmd_Init` that only registers the built-in commands provided by TclX. However, gaining access to the Tcl procedures added by tclX is awkward because tclX insists on completely overriding the standard tcl library. It goes so far as to change the result of the `info library` call if you use its script library mechanism. This means that you can use one library directory or the other, but not both at the same time. You will have to copy the `tcl.tlib` file out of the tclX library directory into another location, or teach your application where to find it. It is probably easiest to use the tclX library system if you are using Extended Tcl.

More UNIX system calls

Extended Tcl provides several UNIX-related commands. Most of the following should be familar to a UNIX programmer: `alarm`, `chgrp`, `chmod`, `chown`, `chroot`, `convertclock`, `dup`, `execl`, `fmtclock`, `fork`, `getclock`, `kill`, `link`, `mkdir`, `nice`, `pipe`, `readdir`, `rmdir`, `select`, `signal`, `sleep`, `system`, `sync`, `times`, `umask`, `unlink`, and `wait`. The `id` command provides several operation on user, group, and process IDs.

File operations

The `bsearch` command does a binary search of a sorted file. The `copyfile` command copies a file, and `frename` changes the name of a file. Low level file controls are provided with `fcntl`, `flock`, `funlock`, and `fstat`. Use `lgets` to read the next complete Tcl list into a list varible. The `read_file` and `write_file` commands provide basic I/O operations. The `recusrive_glob` command matches file names in a directory hierarchy.

New loop constructs

The `loop` command is an optimized version of the `for` loop that works with constant start, end, and increment values. The `for_array_keys` command loops over the contents of an array. The `for_recursive_glob` command loops over file names that match a pattern. The `for_file` command loops over lines in a file.

Command line addons

A script can explicitly enter an interactive command loop with the `commandloop` command. The `echo` command makes it easy to print values. The `dirs`, `pushd`, and `popd` commands provide a stack of working directories. The `infox` command provides information like the application name, the version nubmer, and so on.

Debugging and development support

The `cmdtrace` procedure shows what commands are being executed. The `profile` command sets up a profile of the CPU time used by each procedure or command. The profile results are formatted with the `profrep` command. Use `showprocs` to display a procedure definition, `edprocs` to bring up an editor on a procedure, and `saveprocs` to save procedure definitions to a file.

TCP/IP access

The `server_info` command returns name, address, and alias information about servers. The `server_open` command opens a TCP socket to specified host and port. The `fstat remotehost` command returns the IP address of the remote peer if the file is an open socket. Once a socket is opened, it can be read and written with the regular file I/O commands, and `select` can be used to wait for the socket to be ready for I/O.

File scanning (i.e., awk)

You can search for patterns in files and then execute commands when lines match those patterns. This provides a similar sort of functionality as *awk* . The process starts by defining a context with the `scancontext` command. The `scanmatch` command registers patterns and commands. The `scanfile` command reads a file and does matching according to a context. When a line is matched, information is placed into the `matchInfo` array for use by the associated command.

Math functions as commands

Procedures are defined that let you use the math functions as command names. The commands are implemented like this.

```
proc sin {x} { uplevel [list expr sin($x)] }
```

List operations

New built-in list operations are provided. The `lvarpop` command removes an element from a list and returns its value, which is useful for processing command line arguments. The `lvarpush` command is similar to `linsert`. The `lassign` command assigns a set of variables values from a list. The `lmatch` command returns all the elements of a list that match a pattern. The `lempty` command is a shorthand for testing the list length against zero. The `lvarcat` command is similar to the `lappend` command.

There are four procedures that provide higher level list operations. The `intersect` procedure returns the common elements of two lists. The `intersect3` procedure returns three lists: the elements only in the first list, the elements in both lists, and the elements only in the second list. The `union` procedure merges to lists. The `lrmdups` procedure removes duplicates from a list.

Keyed list data structure

A keyed list is a list where each element is a key-value pair . The value can also be a keyed list, leading to a recursive data structure. Extended Tcl provides built-in support to make accessing keyed lists efficient. The `keylset` command sets the value associated with a key. The `keylkeys` returns a list of the keys in a keyed list. The `keylget` command returns the value associated with a key. The `keyldel` command deletes a key-value pair.

String utilities

Several built-in commands provide the same function as uses of the `string` command. The `cequal` command is short for checking `string compare` with zero. The `clength` command is short for `string length`. The `crange` command is short for `string range`. The `cindex` command is short for `string index`. The `collate` command is short for string compare, plus it has locale support for different character sets. Because these are built-in commands, they are faster that writing Tcl procedures to obtain the shorthand, and a tiny bit faster than the `string` command because there is less argument checking.

The `ctype` command provides several operations on strings, such as checking for spaces, alphanumberics, and digits. It can also convert between characters and their ordinal values.

The `cexpand` command expands backslash sequences in a string. The `replicat` command creates copies of a string. The `translit` command maps characters in a string to new values in a similar fashion as the UNIX *tr* program.

XPG/3 message catalog

The XPG/3 message catalog supports internationalization of your program. You build a catalog that has messages in different languages. The `catopen` command returns a handle on a catalog. The `catgets` command takes a default string, looks for it in the catalog, and returns the right string for the current locale setting. The `catclose` command closes the handle on the catalog.

Memory debugging

Extended Tcl provides both C library hooks to help you debug memory problems, and a Tcl interface that dumps out a map of how your dynamic memory arena is being used. Consult the Memory man page that comes with TclX for details.

Expect: Controlling Interactive Programs

Expect gives you control over interactive programs. For example, you can have two instances of the *chess* program play each other. More practical applications include automated access to FTP sites or navigation through network firewalls. If you are stuck with a program that does something useful but insists on an interactive interface, then you can automate its use with expect. It provides sophisticated control over processes and UNIX pseudo-terminals, so you can do things with expect that you just cannot do with ordinary shell scripts.

The *expect* shell program includes the core Tcl commands and the additional expect commands. The *expectk* shell also includes Tk, so you can have a graphical interface. If you have a custom C program you can include the expect commands by linking in its C library, `libexpect.a`. You can use the C interface directly, but in nearly all cases you will find it easier to drive expect (and the rest of your application) from Tcl.

The expect package was designed and implemented by Don Libes. Historically it is the first extension package. Libes wrote the intial version in about two weeks after he first heard about Tcl. He had long wanted to write something like expect, and Tcl provided just the infrastructure that he needed to get started. By now the expect package is quite sophisticated. Libes has an excellent book about Expect, *Exploring Expect*, published by O'Reilly & Associates, Inc.

As of this writing, the current version of expect is 5.13, and it is compatible with Tcl 7.4. A version 5.14 is expected which will take advantage of some of the new features in Tcl and improve the debugger that comes with expect. You can always fetch the latest version of expect by FTP from the following site and file name.

```
ftp.cme.nist.gov:/pub/expect/expect.tar.Z
```

The rest of this section provides a short overview of expect and gives a few tips that may help you understand how expect works. Expect is a rich facility, however, and this section only scratches the surface.

The core expect commandsl

There are four fundamental commands added by expect: `spawn`, `exp_send`, `expect`, and `interact`. The `spawn` command executes a program and returns a handle that is used to control I/O to the program. The `exp_send` command sends input to the program. (If you are not also using Tk, then you can shorten this command to `send`.) The `expect` command pattern matches on output from the program. The `expect` command is used somewhat like the Tcl `switch` command. There are several branches that have different patterns, and a block fo Tcl commands is associated with each pattern. When the program generates output that matches a pattern, the associated Tcl commands are executed.

The `send_user` and `expect_user` commands are analagous to `exp_send` and `expect`, but they use the I/O connection to the user instead of the process. A common idiom is to `expect` a prompt from the process, `expect_user` the response, and then `exp_send` the response to the program. Generally the user sees everything so you do not need to `send_user` all the program output.

The `interact` command reconnects the program and the user so you can interact with the program directly. The `interact` command also does pattern matching, so you can set up Tcl commands to execute when you type certain character sequences or when the program emits certain strings. Thus you can switch back and forth between human interaction and program controlled interaction. Expect is quite powerful!

Pattern matching

The default pattern matching used by expect is *glob*-style. You can use regular expression matching by specifying the `-re` option to the `expect` command. Most of the work in writing an expect script is getting the patterns right.When writing your patterns, it is important to remember that expect relies on the Tcl parser to expand backslash sequences like `\r\n` (carriage return, newline), which is often an important part of an expect pattern. There is often a `\n` and or `\r` at the end of a pattern to make sure that a whole line is matched, for example. You need to group your patterns with double-quotes, not braces, to alllow backslash substitutions.

If you use regular expressions the quoting can get complicated. You have to worry about square brackets and dollar signs, which have different meanings to Tcl and the regular expression parser. Matching a literal backslash is the most tedious because it is special to both Tcl and the regular expression parser. You'll need four backslashes, which Tcl maps into two, which the regular expression interprets as a single literal backslash.

There are a few pattern keywords. If an expect does not match within the timeout period, the `timeout` pattern is matched. If the process closes its output stream, then the `eof` pattern is matched.

Important variables

Expect uses a number of variables. A few of the more commonly used variables are described here.

The `spawn` command returns a value that is also placed into the `spawn_id` variable. If you spawn several programs, you can implement a sort of job control by changing the value of the global `spawn_id` variable. This affects which process is involved with `exp_send`, `expect`, and `interact` commands. You can also specify the id explicity with a `-i` argument to those commands.

Hint: If you use `spawn` in a procedure, you probably need to declare `spawn_id` as a `global` variable. Otherwise, an `exp_send` or `expect` in anther context will not see the right value for `spawn_id`. It is not strictly necessary to make `spawn_id` global, but it is certainly necessary if you use it in different contexts.

The `timeout` variable controls how long expect waits for a match. Its value is in seconds.

When a pattern is matched by expect, the results are put into the `expect_out` array. The `expect_out(0,string)` element has the part of the input that matched the pattern. If you use subpatterns in your regular expressions, the parts that match those are available in `expect_out(1,string)`, `expect_out(2,string)`, and so on. The `expect_out(buffer)` element has the input that matched the pattern, plus everything read before the match since the last `expect` match.The `interact` command initializes an array called `interact_out` which has a similar structure.

The log_user variable controls whether or not the user sees the output from the process. For some programs you may want to supress all the output. In other cases it may be important for the user to see what is happening so they know when to type responses, such as passwords.

An example expect script

The following example demonstrates the usefulness of expect. The situation is an FTP server that uses a challenge response security system. Without expect, the user needs two windows. In one window they run FTP. In the other window they run a program that computes the reponse to the challenge from FTP. They use cut and paste to feed the challenge from FTP to the key program, and use it again to feed the response back to FTP. It is a tedious task that can be fully automated with expect.

Example 32–1 A sample expect script.

```
#!/usr/local/bin/expect -f
# This logs into the FTP machine and
# handles the S/Key authentication dance.

# Setup global timeout action. Any expect that does not match
# in timeout seconds will trigger this action.
expect_after timeout {
    send_user "Timeout waiting for response\n"
    exit 1
}
set timeout 30   ;# seconds

# Run ftp and wait for Name prompt
spawn ftp parcftp.xerox.com
expect {*Name *:}

# Get the name from the user pass it to FTP
expect_user "*\n"
exp_send $expect_out(buffer)

# Wait for Skey Challenge, which looks like:
# 331 Skey Challenge "s/key 664 be42066"
expect -re {331.*s/key ([^"]+)"} {
    set skey $expect_out(1,string)
}
# Save the spawn ID of ftp and then
# run the key program with the challenge as the argument
set ftpid $spawn_id
eval {spawn key} $skey

# Read password with no echoing, pass it to key
system stty -echo
expect {password:}
expect_user "*\n" { send_user \n }
exp_send $expect_out(buffer)
```

```
# Wait for the key response
expect -re "\n(.+)\[\r\n\]" {
    set response $expect_out(1,string)
}
# Close down the connection to the key program
close
system stty echo

# Pull ftp back into the foreground
set spawn_id $ftpid
system stty echo
exp_send $response\n

# Interact with FTP normally
expect {*ftp>*} { interact }
```

The example uses the `expect_after` command to set up a global timeout action. The alternative is to include a timeout pattern on each expect command. In that case the commands would look like this:

```
expect {*Name *:} { # Do nothing } \
    timeout { send_user "You have to login!\n" ; exit 1 }
```

The `system` command is used to run UNIX programs. It is like exec, except that the output is not returned and a /bin/sh is used to interpret the command. The `stty echo` command turns off echoing on the terminal so the user doesn't see their password being typed.

Debugging expect scripts

The *expect* shell takes a `-d` flag that turns on debugging output. This shows you all the characters generated by a program and all the attempts at pattern matching. This is very useful. You become very aware of little details. Remember that programs actually generate `\r\n` at the end of a line, even though their `printf` only includes `\n`. The terminal driver converts a simple newline into a carrage return, line feed sequence. Conversely, when you send data to a program, you need to explicitly include a `\n`, but you don't send `\r`.

Expect includes a debugger, which is described in the next section and in Chapter 8. If you specify the `-D 1` command line argument to the *expect* shell, then this debugger is entered before your script starts execution. If you specify the `-D 0` command line argument, then the debugger is entered if you generate a keyboard interrupt (`SIGINT`).

Expect's Tcl debugger

The expect package includes a Tcl debugger. It lets you set breakpoints and look at the Tcl execution stack.This section explains what you need to add to your C program to make the debugger available to scripts. The interactive use of the debugger is described in Chapter 8 on page 84.

The Dbg C interface

The debugger is implemented in one file, `Dbg.c`, that is part of the expect library. You can make the debugger separately from expect, but it is easiest to link against the expect library. The core procedures are `Dbg_On` and `Dbg_Off`.*

```
void *Dbg_On(Tcl_Interp *interp, int immediate);
void *Dbg_Off(Tcl_Interp *interp);
```

If `immediate` is 1, then `Dbg_On` enters an interactive command loop right away. Otherwise the debugger waits until just before the next command is evaluated. It is reasonable to call `Dbg_On` with `immediate` set to zero from inside a `SIGINT` interrupt handler.

The `Dbg_ArgcArgv` call lets the debugger make a copy of the command line arguments. It wants to print this information as part of its call stack display. If the *copy* argument is 1, a copy of the argument strings is made and a pointer to the allocated memory is returned. Otherwise it just retains a pointer and returns 0. The copy may be necessary because the `Tk_ParseArgv` procedure will modify the argument list. Call `Dbg_ArgcArgv` first.

```
char **Dbg_ArgcArgv(int argc, char *argv[], int copy);
```

The `Dbg_Active` procedure returns 1 if the debugger is currently on. It does no harm, by the way, to call `Dbg_On` if the debugger is already active.

```
int Dbg_Active(Tcl_Interp *interp);
```

The remaining procedures are only needed if you want to refine the behavior of the debugger. You can change the command interpreter, and you can filter out commands so the debugger ignores them.

The `Dbg_Interactor` procedure registers a command loop implementation and a clientdata pointer. It returns the previously registered procedure.

```
Dbg_InterProc
Dbg_Interactor(Tcl_Interp *interp,
     Dbg_InterpProc *inter_proc, ClientData data);
```

The command loop procedure needs to have the following signature.

```
int myinteractor(Tcl_Interp *interp);
```

Look in the Dbg.c file at the simpler_interactor procedure to see how the command loop works. In practice the default interactor is just fine.

The Dbg_IgnoreFuncs procedure registers a filtering function that decides what Tcl commands should be ignored. It returns the previously registered filter procedure. The filter should be relatively efficient because it is called before every command when the debugger is enabled.

```
Dbg_IgnoreFuncsProc
Dbg_IgnoreFuncs(Tcl_Interp *interp,
     Dbg_IgnoreFuncsProc *ignoreproc);
```

* I will give the C signatures for the procedures involved because I no longer see them in the standard Expect documentation. Libes described the debugger in a nice little paper, "A Debugger for Tcl Applications", that appeared in the 1993 Tcl/Tk workshop.

The `ignoreproc` procedure just takes a string as an argument, which is the name of the command about to be executed. It returns 1 if the command should be ignored.

```
int ignoreproc(char *s);
```

Handling SIGINT

A common way to enter the debugger is in response to a keyboard interrupt. The details of signal handling vary a little from system to system, so you may have to adjust this code somewhat. The `Sig_Setup` procedure is meant to be called early in your main program. It does two things. It registers a signal handler, and it registers a Tcl asynchronous event. It isn't safe to do much more than set a variable value inside a signal handler, and it certainly is not safe to call `Tcl_Eval` in a signal handler. However, the Tcl interpreter lets you register procedures to be called at a safe point. The registration is done with `Tcl_AsyncCreate`, and the handler is enabled with `Tcl_AsyncMark`. Finally, within the async handler the debugger is entered by calling `Dbg_On`.

Example 32–2 A SIGINT handler.

```
#include <signal.h>
/*
 * Token and handler procedure for async event.
 */
Tcl_AsyncHandler sig_Token;
int Sig_HandleSafe(ClientData data,
    Tcl_Interp *interp, int code);
/*
 * Set up a signal handler for interrupts.
 * This also registers a handler for a Tcl asynchronous
 * event, which is enabled in the interrupt handler.
 */
void
Sig_Setup(interp)
    Tcl_Interp *interp;
{
    RETSIGTYPE (*oldhandler)();
    oldhandler = signal(SIGINT, Sig_HandleINT);
    if ((int)oldhandler == -1) {
        perror("signal failed");
        exit(1);
    }
    sig_Token = Tcl_AsyncCreate(Sig_HandleSafe, NULL);
#if !defined(__hpux) & !defined(SVR4)
    /*
     * Ensure that wait() kicks out on interrupt.
     */
    siginterrupt(SIGINT, 1);on interrupt */
#endif
}
/*
```

```
 * Invoked upon interrupt (control-C)
 */
RETSIGTYPE
Sig_HandleINT(sig, code, scp, addr)
    int sig, code;
    struct sigcontext *scp;
    char *addr;
{
    Tcl_AsyncMark(sig_Token);
}
/*
 * Invoked at a safe point sometime after Tcl_AsyncMark
 */
int
Sig_HandleSafe(data, interp, code)
    ClientData data;
    Tcl_Interp *interp;
    int code;
{
    Dbg_On(interp, 1);/* Enter the Tcl debugger */
}
```

BLT

The BLT package has a number of Tk extensions: a bar graph widget, an X-Y graph widget, a drag-and-drop facility, a table geometry manager, a busy window, and more. This section provides an overview of this excellect collection of extensions. The gadgets in BLT where designed and built by George Howlett, and Michael McLennan built the drag and drop facility. As of this writing BLT version 1.7 is compatible with Tk 3.6 and Tcl 7.4. A 1.8 (or 2.0) release is expected shortly that will be compatible with Tk 4.0 and Tcl 7.4. You can find the BLT package in the Tcl archives in the extensions directory.

`ftp.aud.alcaltel.com:pub/tcl/extensions/BLT-1.7.tar.gz`

The BLT package is very clean to add to your application. All the commands and variables that it defines begin with the `blt_` prefix. Initialization simply requires calling `Blt_Init` in your `Tcl_AppInit` procedure.

Drag and drop

The drag and drop paradigm lets you "pick up" an object in one window and drag it into another window, even if that window is in another application. The `blt_drag&drop` command provides a drag and drop cability for Tk applications. A right click in a window creates a token window that you drag into another window. When the object is released, the TK send command is used to communicate between the sender and the receiver.

Hypertext

The blt_htext widget combines text and other widgets in the same scrollable window. The widget lets you embed Tcl commands in the text, and these commands are invoked as the text is parsed and displayed. Tk 4.0 added similar functions to the standard Tk text widget, except for the embedded Tcl commands in the text.

Graphs

The blt_graph widget provides and X-Y plotting widget. You can configure the graph in a variety of ways. The elements of the graph can have tags that are similar to the canvas widget object tags. The blt_barchart widget provides a bar graph display. It also has a tag facility.

Table geometry manager

The table geometry manager lets you position windows on a grid, which makes it easy to line things up. The interface is designed by defining a grid that can uneven spacing for the rows and colums. Widgets are positioned by specifying a grid location and the number of rows and columns the widget can span. You can constrain the size of the widget in various ways. A table geometry manager will probably be added to the standard Tk library in a future release.

Bitmap support

In standard Tk, the only way to define new bitmaps in Tcl is to specify a file that contains its definition. The blt_bitmap command lets you define a new bitmap and give it a symbolic name. It also lets you query the names and sizes of existing bitmaps.

Background exec

The blt_bgexec command runs a pipeline of processes in the background. The output is collected into a Tcl variable. You use tkwait variable to detect when the pipeline has completed operation.

Busy window

The blt_busy command creates an invisible window that covers your application, provides a different cursor, and prevents the user from interacting with your application.

Tracing Tcl commands

The blt_watch provides a Tcl interface to the Tcl trace facility. It lets you register Tcl commands to be called before and after the execution of all commands. You can implement logging and profiling with this facility. The `blt_debug` command displays each Tcl command before and after substitutions are performed.

The old-fashioned cutbuffer

Old versions of X programs may still use the out-dated cutbuffer facility. The `blt_cutbuffer` command provides an interface to the cutbuffer. This can help improve your interoperation with other tools, although you should use the regular selection mechanism by default.

Tcl-DP

The Tcl-DP extension creates network connections between Tcl interpreters using the TCP protocol. It provides a client-server model so you can execute Tcl commands in other interpreters. This is similar to the Tk `send` command, except that Tcl-DP is not limited to applications on the same display. You can have Tcl-DP without Tk, too, for clients or servers that have no graphical interface.

There are three shell programs: *dpwish* includes Tk and Tcl-DP. *dptcl* just has the Tk event loop and Tcl-DP. The *dpsh* shell includes Tk, but can be started with a `-notk` argument to prevent it from opening a display. Of course, all of these include the standard Tcl commands, too.

The low-level networking functions are exported to both Tcl and C. For example, the `dp_packetSend` Tcl `command` sends a network packet. The same function is available from C with the `Tdp_PacketSend` procedure. Other C procedures include `Tdp_FindAddress`, `Tdp_CreateAddress`, `Tdp_PacketReceive`, and `Tdp_RPC`. These are bundled into the `libdpnetwork.a` library archive.

Tcl-DP was designed and built by Brian Smith, Steve Yen, and Stephen Tu. Version 3.2 is compatible with Tcl 7.3 and Tk 3.6. When it is released, version 3.3 will be compatible with Tcl 7.4 and Tk 4.0. You can find the Tcl-DP distribution at the following FTP site.

```
mm-ftp.cs.berkeley.edu
/pub/multimedia/Tcl-DP/tcl-dp3.2.tar.Z
```

Remote Procedure Call

The `dp_MakeRPCServer` command sets up the server's network socket. The `dp_MakeRPCClient` command sets up the client and connects to the server. The `dp_RPC` command invokes a Tcl command in the server. The `do_RDO` command is similar, except that it does not wait for completion of the command. It takes an optional callback command so you can get notified when the asynchronous operation completes. The `dp_CancelRPC` is used to cancel asynchronous RPCs. The `dp_CloseRPC` shuts down one end of a connection.

Servers are identified by a network address and port number. No higher-level name service is provided, although you can often do quite well by using a file in your shared network file system.

A simple form of security is provided. A server can define a set of trusted hosts with the `dp_Host` command. Connections will only be accepted from clients on those trusted hosts. Each command can be verified before execution. The `dp_SetCheckCmd` registers a procedure that is called to verify a client request. You could use the verification hook to enforce an authentication dialog, although the TCP connect is not encrypted.

Connection setup

The `dp_connect -server` command is used by servers to create a listening socket. Servers then use the `dp_accept` command to wait for new connections from clients. Clients use the `dp_connect` command (without `-server`) to connect to a server . The connect uses TCP by default. The `-udp` option creates a UDP connection. The `dp_socketOption` provides an interface to the `setsockopt` and `getsockopt` system calls. The `dp_shutdown` command is used to close down a connection. The dp_atclose command registers a command to be called just before a connection is closed. The `dp_atexit` command registers a command to be called when the process is exiting. These commands are used by the RPC-related procedures described above.

Sending network data

The regular `puts` and gets commands can be used to transfer line-oriented data over a connection.If you use these commands, it is necessary to use the `eof` command to detect a closed connection.

The `dp_send` and `dp_receive` commands transfer data across a connection. The `dp_packetSend` and `dp_packetReceive` transfer blocks of data while preserving message boundaries. These use the TCP protocol, and they automatically handle closed connections.

Using UDP

The `dp_sendTo` and `dp_receiveFrom` commands transfer packets using the UDP protocol. These commands take an argument that specifies the remote network address with which to communicate. These arguments are returned by `dp_address` command, which maintains an address table.

Event processing

Tcl-DP uses the Tk event loop mechanism to wait for network data. It provides several commands that relate to the event loop processing. The dp_filehandler command registers a command to be called when data is ready on a connection. The dp_isready command indicates if data is available on a connection. The dp_whenidle command schedules a command to occur at the next idle point.

The following commands are provided by Tcl-DP in case Tk is not available. The `dp_update` command forces a trip through the event loop. The `dp_after` command executes a command after a specified time interval. `dp_waitvariable` waits for a variable to be modified. These are equivalent to the following Tk commands: `update`, `after`, and `tkwait variable`.

Replicated objects

A simple replicated object package is built on top of Tcl-DP. In the model, an object is procedure with methods and slot values. Every object must implement the `configure`, `slotvalue`, and `destroy` methods. The `configure` method is used to query and set slot values, much like the Tk widget `configure` operation. The `slotvalue` method returns the value of a slot, much like the `cget` operation of a Tk widget.

An object is replicated on one or more sites, and updates to an object are reflected in all copies. You can register callbacks that are invoked when the object is modified. The `dp_setf` command sets a replicated slot value. The `dp_getf` command returns a slot value. The `dp_DistributeObject` command arranges for an object to be replicated on one or more sites. The `dp_UndistributeObject` breaks the shared relationship among distributed objects. The `dp_SetTrigger`, `dp_AppendTrigger`, and `dp_AppendTriggerUnique` commands register a Tcl command to be called with a slot is modified. The `dp_GetTriggers` command returns the registered triggers. The `dp_ReleaseTrigger` removes one trigger, and the `dp_ClearTrigger` removes all triggers from a slot.

An object can be implemented by a command procedure written in C. All it has to do is adhear to the conventions outlined above about what methods it supports. The method is just the first argument to the procedure. The object will be invoked as follows:

```
objName method ?args?
```

Obviously, there must also be a command that creates instances of the object. This should have the following form:

```
makeCmd objName ?-slot value? ?-slot value? ...
```

You can also implement objects with Tcl procedures. Several commands are provided to support this: `dp_objectCreateProc`, `dp_objectExists`, `dp_objectFree`, `dp_objectConfigure` `dp_objectSlot`, `dp_objectSlotSet`, `dp_objectSlotAppend`, and `dp_objectSlots`.

The [incr tcl] Object System

The [incr tcl] extension provides an object system for Tcl. Its funny name is an obvious spoof on C++. This extension adds classes with multiple inheritence to Tcl. A class has methods, class procedures, private variables, public variables, and class variables. All of these items are contained within their own scope. The class procedures and class variables are shared by all objects in a class. The methods, private variables, and public variables are per-object. A public variable is a lot like the attributes for Tk widgets. Its value is defined when the object is created, or changed later with a `config` method. The `config` syntax looks like the syntax used with Tk widgets and their attributes.

The following summary of the [incr tcl] commands is taken straight from the man page for the package.

Example 32–3 Summary of [incr tcl] commands

```
itcl_class className {
    inherit baseClass ?baseClass...?

    constructor args body
    destructor body
    # A method is per-object
    method name args body
    # proc creates class procedures
    proc name args body

    # public vars have a config syntax to set their value
    public varName ?init? ?config?
    # protected variables are per-object
    protected varName ?init?
    # common variables are shared by the whole class
    common varName ?init?
}

# Create an object. The second form chooses the name.
className objName ?args...?
className #auto ?args...?

# Invoke a class procedure proc from the global scope
className :: proc ?args...?

# Invoke an object method
objName method ?args...?

# Built-in methods
objName isa className
objName delete
objName info option ?args?

# Get info about classes and objects
itcl_info classes ?pattern?
itcl_info objects ?pattern? ?-class className? ?-isa
className?

# Commands available within class methods/procs:
global varName ?varName...?
# Run command in the scope of the parent class (up)
previous command ?args...?
# Run command in the scope of the most-specific class (down)
```

```
virtual command ?args...?
```

The most important contribution of [incr tcl] is the added scope control. The elements of a class are hidden inside its scope, so they do not clutter the global Tcl name space. When you are inside a method or class procedure, you can directly name other methods, class procedures, and the various class variables. When you are outside a class, you can only invoke its methods through an object. It is also possible to access class procedures with the :: syntax shown in the previous example. The scope control is implemented by creating a new Tcl interpreter for each class, although this is hidden from you when you use the extension.

There is one restriction on the multiple inheritence provided by [incr tcl]. The inheritence graph must be a tree, not a more general directed-acyclic-graph. For example, if you have a very general class called `Obj` that two classes `A` and `B` inherit, then class `C` cannot inherit from both `A` and `B`. That causes the elements of `Obj` to be inherited by two paths, and the implementation of [incr tcl] does not allow this. You would have to replicate the elements of `Obj` in `A` and `B` in this example.

Tcl_AppInit With Extensions

The next example shows a `Tcl_AppInit` that initializes several packages in addition to the new commands and widgets from the previous chapters. Most of the packages available from the Tcl archive have been structured so you can initialize them with a single call to there *Package*`_Init` procedure. To create the full application, the `Tcl_AppInit` routine is linked with the libraries for Tcl and the various extensions being used. The Makefile for that is given in the next example.

Example 32–4 Tcl_AppInit and extension packages.

```
/* supertcl.c */
#include <stdio.h>
#include <tk.h>
#include <tclExtend.h>

extern char *exp_argv0;/* For expect */

/*
 * Our clock widget.
 */
int ClockCmd(ClientData clientData,
             Tcl_Interp *interp,
             int argc, char *argv[]);
/*
 * Our pixmap image type.
 */
extern Tk_ImageType tkPixmapImageType;
```

```
main(int argc, char *argv[]) {
    /*
     * Save arguments for expect and its debugger.
     */
    exp_argv0 = argv[0];/* Needed by expect */
    Dbg_ArgcArgv(argc, argv, 1);
    /*
     * Create the main window. This calls
     * back into Tcl_AppInit.
     */
    Tk_Main(argc, argv);
    exit(0);
}

int
Tcl_AppInit(Tcl_Interp *interp) {
    char *value;
    Tk_Window main = Tk_MainWindow(interp);
    /*
     * Initialize extensions
    */
    if (TclX_Init(interp) == TCL_ERROR) {
        /* TclX_Init is called instead of Tcl_Init */
        return TCL_ERROR;
    }
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tdp_Init(interp) == TCL_ERROR) {/* Tcl-DP */
        return TCL_ERROR;
    }
    if (Blt_Init(interp) == TCL_ERROR) {/* BLT */
        return TCL_ERROR;
    }
    if (Exp_Init(interp) == TCL_ERROR) {/* Expect */
        return TCL_ERROR;
    }
    /*
     * This affects X resource names.
     */
    Tk_SetClass(main, "SuperTcl");
    /*
     * Our own extra commands.
     */
    Tcl_CreateCommand(interp, "clock", ClockCmd,
        (ClientData)Tk_MainWindow(interp),
        (Tcl_CmdDeleteProc *)NULL);

    Tk_CreateImageType(&tkPixmapImageType);
    /*
     * The remaining lines are similar to code in TkX_Init.
     * The tclApp variables define info returned by infox
     * The Tcl_SetupSigInt is a TclX utility that lets
     * keyboard interrupts stop the current Tcl command.
```

```
     */
    tclAppName = "SuperTcl";
    tclAppLongname = "Tcl-Tk-TclX-DP-BLT-Expect-[incr tcl]";
    tclAppVersion = "1.0";
    /*
    * If we are going to be interactive,
    * Setup SIGINT handling.
    */
    value = Tcl_GetVar (interp, "tcl_interactive",
        TCL_GLOBAL_ONLY);
    if ((value != NULL) && (value [0] != '0'))
        Tcl_SetupSigInt ();

    return TCL_OK;
}
```

Because Extended Tcl is being used, `TclX_Init` is called instead of `Tcl_Init`. I think this is an unfortunate inconsistency, but Extended Tcl insists on duplicating some things in Tcl and Tk, so this is necessary to avoid linker problems. I have rebelled a little bit by calling `Tk_Init` instead of `TkX_Init`, which is recommended by the TclX documentation. However, this means I do much the same work by calling `Tcl_SetupSigInt` and defining the various `tclApp` varibles.

The Makefile for the superwish program is given in the next example. The program uses a mixture of shared and static libraries. Ideally all the packages can be set up as shared libraries in order to reduce the size of the shell programs. Shared libraries are described in LIBRARY chapter.

Example 32–5 Makefile for supertcl.

```
# At our site all the Tcl packages have their
# libraries in /import/tcl/lib, and the files
# have the version number in them explicitly
# The .so files are shared libraries

TCL_LIB = /import/tcl/lib/libtcl7_4_g.a
TK_LIB = /import/tcl/lib/libtk4_0_g.a

BLT_LIB = /import/tcl/lib/libBLT1_7.a
DP_LIB = /import/tcl/lib/libdpnetwork.so.3.2
EXP_LIB = /import/tcl/lib/libexpect5_13.a
TCLX_LIB = /import/tcl/lib/libtclx7_4.a
TKX_LIB = /import/tcl/lib/libtkx4_0.a
INCR_LIB = /import/tcl/lib/libitcl.so.1.3

# The include files are also organized under
# /import/tcl/include in directories that
# reflect the packages' version numbers.
INCS = -I/import/tcl/include/tk4.0 \
    -I/import/tcl/include/tclX7.4a \
    -I/import/tcl/include/tcl7.4 \
    -I/import/X11R4/usr/include
```

```
CFLAGS = -g $(INCS)
CC = gcc

# The order of these libraries is important, expecially
# having TKX TK TCLX TCL come last.
# Your site may not need -lm for the math library
ALL_LIBS = $(DP_LIB) $(EXP_LIB) $(BLT_LIB) $(INCR_LIB) \
	$(TKX_LIB) $(TK_LIB) $(TCLX_LIB) $(TCL_LIB) -lX11 -lm

OBJS = supertcl.o tkWidget.o tkImgPixmap.o
supertcl: $(OBJS)
	$(CC) -o supertcl $(TRACE) $(OBJS) $(ALL_LIBS)
```

Other Extensions

There are lots of contributed Tcl packages. You should check out the Tcl FTP archive site, which is currently

ftp.aud.alcatel.com

There is an excellent set of Frequently Asked Questions files that are maintained by Larry Virden. Volumes 4 and 5 of the FAQ list the contributed extensions and the contributed applications, respectively. The TIX package provides many compound widgets and an object system that is designed to support them. The jstools package, which is contributed by Jay Sekora, contains a collection of useful support scripts and applications. The list goes on and on, and gets updated with new contributes regularly

Tcl applications

You should try out my nifty mail reader, *exmh*, which provides a graphical front end for MH mail. It supports the MIME standard for multimedia mail, and the PGP tool for encryption and digital signatures. The *tkman* program, written by Tom Phelps, provides a great user interface to the UNIX man pages. The *ical* program is a very useful calendar manager. There are several Tcl-based editors. I ported Ousterhout's mx editor into the Tk framework, and the result is *mxedit*. If you like interface builders, try out *XF*, which lets you build Tk interfaces interactively. Again, the list goes on and on

CHAPTER 33

Porting to Tk 4.0

This chapter has notes about upgrading your application from earlier versions of Tk such as Tk 3.6. This includes notable new features that you may want to take advantage of as well as things that need to be fixed because of incompatible changes.

Porting your scripts from any of the Tk 3.* releases is pretty easy. Not that many things have changed. The sections in this chapter summarize what has changed in Tk 4.0 and what some of the new commands are.

wish

The *wish* shell no longer requires a `-file` (or `-f`) argument, so you can drop this from your script header lines. This flag is still valid, but no longer necessary.

The class name of the application is set from the name of the script file instead of always being `Tk`. If the script is `/usr/local/bin/foobar`, then the class is set to `Foobar`, for example.

Obsolete Features

Several features that were replaced in previous versions are now completely unsupported.

The variable that contains the version number is `tk_version`. The ancient (pre 1.4) `tkVersion` is no longer supported.

Button widgets no longer have `activate` and `deactivate` operations.

Instead,configure their `state` attribute.

Menus no longer have `enable` and `disable` operations. Instead, configure their `state` attribute.

The cget Operation

All widgets support a `cget` operation that returns the current value of the specified configuration option. The following two commands are equivalent.

```
lindex [$w config option] 4
$w cget option
```

Nothing breaks with this change, but you should enjoy this feature.

Input Focus Highlight

Each widget can have an input focus highlight, which is a border that is drawn in color when the widget has the input focus. This border is outside the border used to draw the 3D relief for widgets. It has the pleasant visual effect of providing a little bit of space around widgets, even when they do not have the input focus. The addition of the input focus highlight does not break anything, but it will change the appearance of your interfaces a little. See Chapter 22 for a description of the generic widget attributes related to this feature.

Bindings

The hierarchy of bindings has been fixed so that it is actually useful to define bindings at each of the global (i.e., `all`), class, and instance levels. The new `bindtags` command is used to define the order among these sources of binding information. You can also introduce new binding classes, e.g. `InsertMode`, and bind things to that class. Use the `bindtags` command to insert this class into the binding hierarchy. The order of binding classes in the bindtags command determines the order in which bindings are triggered. Use `break` in a binding command to stop the progression, or use `continue` to go on to the next level.

```
bindtags $w [list all Text InsertMode $w]
```

The various `Request` events have gone away: `CirculateRequest`, `ConfigureRequest`, `MapRequest`, `ResizeRequest`.

Extra modifier keys are ignored when matching events. While you can still use the `Any` wild card modifier, it is no longer necessary. The `Alt` and `Meta` modifiers are set up in general way so they are associated with the `Alt_L`, `Alt_R`, `Meta_L`, and `Meta_R` keysyms.

Scrollbar Interface

The interface between scrollbars and the scrollable widgets has changed. Happily the change is transparent to most scripts. If you hook your scrollbars to widgets in the straight-forward way, the new interface is compatible. If you use the `xview` and `yview` widget commands directly, however, you may have to modify your code. The old use still works, but there are new features of the these operations that give you even better control over. You can also query the view state so you do not have to watch the scroll set commands to keep track of what is going on. Finally, scrollable widgets are constrained so that end of their data remains stuck at the bottom (right) of their display.

In most cases, nothing is broken by this change.

Pack info

Version 3 of Tk introduced a new syntax for the `pack` command, but the old syntax was still supported. This continues to be true in nearly all cases except the `pack info` command. If you are still using the old packer format, you should probably take this opportunity to convert to the new packer syntax.

The problem with `pack info` is that its semantics changed. The new operation used to be known as `pack newinfo`. In the old packer, `pack info` returned a list of all the slaves of a window and their packing configuration. Now `pack info` returns the packing configuration for a particular slave. You must first use the `pack slaves` command to get the list of all the slaves, and then use the (new) `pack info` to get their configuration information.

Focus

The focus mechanism has been cleaned up to support different focus windows on different screens. The focus command now takes a `-displayof` argument because of this. Tk now remembers which widget inside each toplevel has the focus. When the focus is given to a toplevel by the window manager, Tk automatically assigns focus to the right widget. The `-lastfor` argument is used to query which widget in a toplevel will get the focus by this means.

The `focus default` and `focus none` commands are no longer supported. There is no real need for `focus default` anymore, and `focus none` can be achieved by passing an empty string to the regular `focus` command.

The `tk_focusFollowsMouse` procedure can be used to change from the default explicit focus model where a widget must claim the focus to one in which moving the mouse into a widget automatically gives it the focus.

The `tk_focusNext` and `tk_focusPrev` procedures are used for keyboard traversal of the focus among widgets. Most widgets have bindings for `<Tab>` and `<Shift-Tab>` that cycle the focus among widgets.

Send

The `send` command has been changed so that it does not time out after 5 seconds, but instead waits indefinitely for a response. Specify the `-async` option if you do not want to wait for a result. You can also specify an alternate display with the `-displayof` option.

The name of an application can be set and queried with the new `tk appname` command. Use this instead of `winfo name "."`.

Because of the changes in the send implementation, it is not possible to use `send` between Tk 4.0 applications and earlier versions.

Internal Button Padding

Buttons and labels have new defaults for the amount of padding around their text. There is more padding now, so your buttons will get bigger if you use the default `padX` and `padY` attributes. The old defaults were one pixel for both attributes. The new defaults are `3m` for `padX` and `1m` for `padY`, which map into three pixels and ten pixels on my display.

There is a difference between buttons and the other button-like widgets. An extra 2 pixels of padding is added, in spite of all padX and padY settings in the case of simple buttons. If you want your checkbuttons, radiobuttons, menubuttons, and buttons all the same dimensions, you'll need two extra pixels of padding for everything but simple buttons.

Radio Buttons

The default value for a radio button is no longer the name of the widget. Instead, it is an empty string. Make sure you specify a `-value` option when setting up your radio buttons.

Entry Widget

The `scrollCommand` attribute changed to `xScrollCommand` to be consistent with other widgets that scroll horizontally. The `view` operation changed to the `xview` operation for the same reasons.

The `delete` operation has changed the meaning of the second index so that the second index refers to the character just after the affected text. The selection operations have changed in a similar fashion. The `sel.last` index refers to the character just after the end of the selection, so deleting from `sel.first` to `sel.last` still works OK. The default bindings have been updated, of course, but if you have custom bindings you will need to fix them.

Menus

The `menu` associated with a `menubutton` must be a child widget of the `menubutton`. Similarly, the `menu` for a cascade menu entry must be a child of the `menu`.

The `@y` index for a menu always returns a valid index, even if the mouse cursor is outside any entry. In this case, it simply returns the index of the closest entry, instead of `none`.

The `selector` attribute is now `selectColor`.

Listboxes

Listboxes have changed quite a bit in Tk 4.0. Chapter 16 has all the details. There are now 4 Motif-like selection styles, and two of these support disjoint selections. The `tk_listboxSingleSelect` procedure no longer exists. Instead, configure the `selectMode` attribute of the listbox.

You can selectively clear the selection, and query if there is a selection in the listbox.

A listbox has an active element, which is drawn with an underline. It is referenced with the `active` index keyword.

The selection commands for listboxes have changed. Change:

```
$listbox select from index1
$listbox select to index2
```

To

```
$listbox select anchor index1
$listbox select set anchor index2
```

The `set` operation takes two indices, and `anchor` is a valid index, which typically corresponds to the start of a selection.

No geometry Attribute

The `frame`, `toplevel`, and `listbox` widgets no longer have a `geometry` attribute. Use the `width` and `height` attributes instead. The `geometry` attribute got confused with geometry specifications for toplevel windows. The use of `width` and `height` is more consistent. Note that for listboxes the width and height is in terms of lines and characters, while for frames and toplevels it is in screen units.

Text Widget

The tags and marks of the text widgets have been cleaned up a bit, justification and spacing is supported, and you can embed widgets in the text display.

A mark now has a gravity, either left or right, that determines what happens when characters are inserted at the mark. With right gravity you get the old behavior: the mark gets pushed along by the inserted text by sticking to the right-hand character . With left gravity it remains stuck. The default is right gravity . The `mark gravity` operation changes it.

When text is inserted, it only picks up tags that are present on both sides of the insert point. Previously it would inherit the tags from the character to the left of the insert mark. You can also override this default behavior by supplying tags to the insert operation.

The widget scan operation supports horizontal scrolling. Instead of using marks like `@y`, you need a mark like `@x,y`.

For a description of the new features, see Chapter 18.

Canvas scrollincrement

The canvas widget no longer has a `scrollIncrement` attribute. Instead, the equivalent of the scroll increment is set at one tenth of the canvas. Scrolling by one page scrolls by nine tenths of the canvas display . (Ugh - I like'd the fine grain control provided by `scrollIncrement`.)

The Selection

The selection support has been generalized in Tk 4.0 to allow use of other selections such as the `CLIPBOARD` and `SECONDARY` selections. The changes to not break anything, but you should check out the new `clipboard` command. Some other toolkits, notably OpenLook, can only paste data from the clipboard.

Color Attributes

The names for some of the color attributes changed.

Table 33–1 Changes in color attribute names

Tk 3.6	Tk4.0
selector	selectColor
Scrollbar .activeForeground	Scrollbar .activeBackground
Scrollbar .background	troughColor
Scrollbar .foreground	Scrollbar .background
Scale.activeForeground	Scale.activeBackground
Scale.background	troughColor
Scale.sliderForeground	Scale.background
(didn' t exist)	highlightColor

The bell Command

The `bell` command can be used to ring the bell associated with the X display. You need to use the *xset* program to modify the parameters of the bell such as volume and duration.

Index

-- 43

A

Abbreviations 92
activeBackground 278, 326
activeForeground 278, 326
after 62, 63, 96
-after 320
anchor 274, 293
-anchor 125, 165, 320
Any modifer 141
APP-DEFAULTS 331
appdefaults 325
APPEND 71
append 21
archive site for Tcl xxviii
args 52, 61, 179, 272, 291
argtest 52
argv 45
array names 39
-aspect 168
aspect ratio 167
aspect ratio of a message widget 270
atime 67
automatic execution of programs 92
auto_noexec 67, 92
auto_path 92, 93

B

B1-Motion 133
background 66, 278, 326
Backslash 14
backslash substitution 6
backslash-newline 167
Backspace 14
beep 176
-before 340
Bell 14
bell, Tk command 176
bg 278
bind 96, 106, 133, 149, 179, 320, 337, 338
    Return key 104
BindSequence 141
bindtags 96, 135
BindYview 318
bitmap label 165
blockSpecial 68
bold 290
-borderwidth 105
borderwidth attribute 105
break 46
break, in bindings 135
button 104, 145, 148, 335
button command 145
button modifiers 140
Button procedure 291
Button-2 133
buttonlist 327
ButtonResources 328

C

call stack, viewing 79
CallTrace 80
Carriage return 14
cascade 330
catch 46, 48, 71, 75, 105, 291, 292, 320, 325, 328, 332, 338
cavity model 118
cd 74, 75
cget 270
Changing a button's command 105
character code 14
characterSpecial 68
checkbutton 145, 149, 330, 337
circles 238
-class 328
classes 324
close 72, 74, 105, 339
close a window 138
-code 49
Color 326
color 278
color name 279
ColorDarken 279
command abbreviation 92
command body 9
command history 92
command line arguments 45
command procedures 342
command substitution 4
CommandEntry 179
comments 13
concat 31, 33, 60, 61, 63, 291
config 105
Configure 138
configure 269, 338
continue 46
continue, in bindings 135
CREAT 71

curly braces 47

D

date 65
default 330
Default parameter values 52
Destroy 139
destroy 338
dialog boxes 167
directory 68
dirname 67
disabledForeground 278, 326
Double 140
double 54
Double quotes 7
double-quotes 47
dp_send 63

E

else 42
elseif 42
entry 104, 148, 177, 337
entrylist 328
env 39
environment variable 39
eof 105
EqualSizedLabels 272
error 48
errorCode 48
-errorcode 49
errorInfo 47, 48
-errorinfo 49
Escape-key 142
eval 59, 60, 61, 92, 179, 272, 292, 330
-exact 43
EXCL 71
exec 66, 69, 321, 339
executable 67
exists 67
expr 4, 9, 10, 20, 42, 52, 54, 68
extension 67

F

fg 278
fifo 68
file dirname 69
file exists 69, 325, 332
file isdirectory 69, 75
file modify times 68
fileeq 69
fileevent 105
-fill 119
FindFile 74
FindFont 291
fixed font 289
FixedWidthLabel 165
flush 72
focus 96, 104
font 324
-font 289, 291
font family 290
Fonts 289
for 46
foreach 39, 45, 52, 73, 75, 321, 336, 339
foreground 278, 326
Form feed 14
format 7, 21, 39, 279
frame 96, 98, 320, 335

G

geometry gridding 110
geometry manager 95
gets 46, 73, 105
glob 24, 74
-glob 43
global 39, 47, 54, 55, 105, 332
goto 46
grouping 2

H

height 270
highlightColor 278

I

if 42
-in 340
incr 9, 30, 42, 55, 56, 336
infinite loop 9
info exists 55, 56, 334
info library 93
init.tcl 93
input focus 106
insertBackground 278
insert, text operation 105
-ipadx 275, 294
isdirectory 67
isfile 67
italic 290
I/O redirection 65

J

join 31
justify 293
-justify 167

K

-keepnewline 66

keysym 137

L

label 104, 148, 165, 336
lappend 31, 32, 46, 59, 93
ldelete 34
length 270
lindex 31, 33, 333
link 68
linsert 31, 34
list 31, 32, 33, 45, 59, 60, 63, 147, 291, 321, 330
llength 31, 33, 39, 333
lower, Tk command 132
lrange 31, 33
lreplace 31, 34, 292
lsearch 31, 34, 292
lsort 31, 39
lstat 67

M

makedir 69
Makefile for a Tcl C program 349
Map 138
mark 108
math expressions 4
MaxLineLength 148
menu 145, 153, 330
menu entries 153
menubutton 145, 153, 330
MenuButtonInner 330
menulist 328
MenuResources 330
message 167, 338
Meta-key 142
mkdir 69
modifier key 138
module 54
mtime 67

N

Newline 14
-nocomplain 74
NOCTTY 71
NONBLOCK 71
null binding 141

O

open 71, 321, 338
    pipeline 105
option 326
option get 328, 330
option menus 159
option readfile 325, 332
orient 271
ovals 238
owned 67

P

pack 106, 148, 320, 335
pack forget 340
pack propagate 272
padX widget attribute 274, 293
padY widget attribute 274, 293
-pageanchor 249
-pagex 249
-pagey 249
Parsing command line arguments 45
pid 54
pipeline 65, 74
place, Tk command 131
points per pixel 290
Pop 57
popup menus 159
POSIX 48
PrefEntrySet 337
PreferencesDialogItem 336
PreferencesReadFile 332
PreferencesReset 339
PreferencesSave 338
Preferences_Add 333, 334, 340
Preferences_Dialog 335
Preferences_Init 332
PrefNukeItemHelp 338
PrefValue 334
PrefValueSet 334
PRIMARY selection 318
PrintByName 56, 151
printenv 39
printf 21
proc 7, 8, 39, 51, 52, 53
Push 56
puts 3, 10, 339
pwd 74

Q

Quit button 104
quoting with backquotes 146

R

radiobuton 330
radiobutton 145, 149, 336
raise 335
raise, Tk command 132
random number 54
randomInit 54
randomRange 54

RDONLY 71
RDWR 71
read 72, 338
readable 68
readlink 68
regexp 26, 142
-regexp 43
regular expressions 25
relative position of windows 130
relative size of windows 130
relief attribute 106
resource 323
return 8, 49, 51, 69
RGB 279
ring the bell 176
rootname 68
round 279

S

scale 271
scan 45
scanf 45
scope 56
Screen units 271
scrollable text lines 211
scrollbar 271
scrollbar, with text 104, 107
ScrollFixup 340
scrolling a frame using a canvas 244
seed 54
see, text operation 105
selectBackground 278, 326
selectColor 278
selectForeground 279
selector 326
send 62, 63
separator 330
set 3, 9, 10, 29
setgrid attribute 107
significant digits 5
size 68, 270
socket 68
sort 72
source 2
split 31, 72, 73, 338
sqrt 8, 55
stack 56
stat 68
Status 332
stderr 3, 71
stdin 3
stdout 3
string 4, 19, 20
string compare 21
string length 39, 46, 330, 336
substitution 2, 3
switch 45, 330
symbolic link 68
syntax 2

T

Tab 14
tail 68
Tcl archive 404
tclIndex 91
Tcl_AppendElement 346
Tcl_AppendResult 346
Tcl_AppInit 344, 356
Tcl_CmdInfo 348
Tcl_CreateCommand 343, 344
Tcl_CreateInterp 343, 359
TCL_ERROR 345
Tcl_Eval 47, 343
Tcl_Eval runs Tcl commands from C 347
Tcl_EvalFile 343
Tcl_GetCommandInfo 349
Tcl_GetInt 345
Tcl_Init 344, 356
Tcl_Invoke bypasses Tcl_Eval 347
TCL_LIBRARY 93
TCL_OK 345
tcl_precision 5
tcl_RcFileName 344, 357
Tcl_SetResult 346
text 211
-textvariable 165
textvariable attribute 104
text, with scrollbar 104, 107
then 42
title bar 105
Title of window 104
tk colormodel 332
tk.tcl 93
Tk_CreateFileHandler 355
Tk_DoOneEvent 355
Tk_LIBRARY 93
tk_library 93
tk_listboxSingleSelect 413
Tk_MainLoop 355
toplevel 335, 338
trace 337
trace variable 340
Triple 140
troughColor 279
TRUNC 71
type 68

U

Unmap 138

unset 57
uplevel 57, 339
upvar 55, 56, 151, 334
User input field, entry 104
user type-in 211

V

variable 3, 30
Vertical tab 14

W

while 9, 45, 73
widgets 95
width 271
window changes size 138
window is opened 138
window manager 105
winfo rgb 279
wm command 105
wm geometry 338
wm title 104, 335
wm transient 338
wrapLength 293
writable 68
WRONLY 71

X

X event 106
X resource database 323
X resource name 270
xfontsel 291
xlsfonts 291