Practical Programming in Tcl and Tk

Brent Welch

DRAFT: Aug 31‘94
(Last tweaked Sep 9 - added list of examples)

This book will be published by
Prentice Hall

Please return marked up copy (before Sept 30) to:

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

The next draft is scheduled for October 15.

Preface

I first heard about Tcl from John Ousterhout in 1988 while I was his PhD 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 does not attempt to duplicate that information, the details of 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. The introductory material presented in Chapters 1 through 8 has some overlap with Ousterhout's book.

How To Review This Book

I am open to comments about all aspects of this book: spelling, awkward sentences, confusing sentences, the Tcl examples, the formatting of the pages, and the overall organization. Just mark up the manuscript with a pen and return it to me. The next one is scheduled for October 15. So, if you could return whatever comments you have before then I would appreciate it. 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.
(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 `harbor.ecn.purdue.edu`. Under the `/pub/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://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://web.cs.ualberta.ca/~wade/Auto/Tcl.html
http://www.sco.com/IXI/of_interest/tcl/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?
```

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.

Chapters 2-4 cover the basic Tcl commands in more detail, including its data types, control flow, procedures and scoping issues. You can skip these chapters if you already know Tcl.

Chapter 5 discusses `eval` and more advanced Tcl coding techniques. If you are running into quoting problems, check out this chapter.

Chapter 6 describes string manipulation and pattern matching. The regular expression capabilities of Tcl are very useful for string processing.

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

Chapter 9 illustrates Tk programming with a number of short examples. The examples include a set of cooperating scripts that use the Tk `send` command for communication between applications.

Chapter 10 explains the packer geometry manager, which is responsible for the layout of widgets on the screen.

Chapter 11 covers event binding. A binding registers a Tcl script that is executed in response to events from the X window system.

Chapter 12 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 13 builds upon Chapter 12 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.

Chapter14 provides a short introduction to using Tcl at the C programming level. It gets you started with integrating Tcl into an existing application.

Outline of Remaining Chapters

Obviously, the book is not yet complete. Here is what I have planned for other chapters.

- Tk text widget. Bindings for editing and keyboard motion. Bindings to drag a selection off the top or bottom of a window and scroll the widget at the same time to allow selection of large chunks. Use of text tags to provide multi-font looks.
- Tk canvas widget. Simple drawing package to demonstrate canvas features.
- Images. Either Tk 4.0 and/or a photo widget. Example of a color scanner interface.
- I/O handling. A UI for a serial-line device like a pen or tablet.

In addition, several chapters that summarize notable extensions are planned.

- Extended Tcl (adds more features)
- Incr Tcl (an object system)
- Expect (for controlling interactive programs)
- Tcl-DP (network socket extensions)
- BLT (table geometry manager, graph widgets, hypertext widget)

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 -f
```

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 `#!` and the trailing `-f`. (For historical reasons, `wish` needs the -f switch to make it read the contents of the file.) 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.

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 6 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! *

* 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.

Example 1–9 Continuing long lines with backslashes.

```
set totalLength [expr [string length $one] + \
       [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 6.

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 4. 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
```

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.

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

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

Table 1–4 Built-in Tcl Commands

`append`	Append arguments to a variable's value. No spaces added.
`array`	Query array state and search through elements.
`break`	Premature loop exit.

Table 1–4 Built-in Tcl Commands

`case`	(Out-of-date - see switch).
`catch`	Trap errors.
`cd`	Change working directory.
`close`	Close an open I/O stream.
`concat`	concatenate arguments with spaces between. Splices lists together.
`continue`	Continue with next loop iteration.
`error`	Raise an error.
`eof`	Check for end-of-file.
`eval`	concatenate arguments and then evaluate them as a command.
`exec`	Fork and execute a UNIX program.
`exit`	Terminate the process.
`expr`	Evaluate a math expression.
`file`	Query the file system.
`flush`	Flush output from an I/O stream's internal buffers.
`for`	Loop construct similar to C `for` statement.
`foreach`	Loop construct over a list of values.
`format`	Format a string similar to C `sprintf`.
`gets`	Read a line of input from an I/O stream.
`glob`	Expand a pattern to matching file names.
`global`	Declare global variables.
`history`	Command-line history control.
`if`	Conditional command. Allows `else` and `elseif` clauses.
`incr`	Increment a variable by an integer amount.
`info`	Query the state of the Tcl interpreter.
`join`	concatenate list elements with a given separator string.
`lappend`	Add elements to the end of a list.
`lindex`	Fetch an element of a list.
`linsert`	Insert elements into a list.
`list`	Create a list out of the arguments.
`llength`	Return the number of elements in a list.
`lrange`	Return a range of list elements.

Table 1–4 Built-in Tcl Commands

`lreplace`	Replace elements of a list
`lsearch`	Search for an element of a list that matches a pattern.
`lsort`	Sort a list.
`open`	Open a file or process pipeline for I/O.
`pid`	Return the process ID.
`proc`	Define a Tcl procedure.
`puts`	Output a string to an I/O stream.
`pwd`	Return the current working directory.
`read`	Read blocks of characters from an I/O stream.
`regexp`	Regular expression matching.
`regsub`	Substitutions based on regular expressions.
`rename`	Change the name of a Tcl command.
`return`	Return a value from a procedure.
`scan`	Similar to the C `sscanf` function.
`seek`	Set the seek offset of an I/O stream.
`set`	Assign a value to a variable.
`source`	Evaluate the Tcl commands in a file.
`split`	Chop a string up into list elements.
`string`	Operate on strings: `compare`, `first`, `index`, `last`, `length`, `match`, `range`, `tolower`, `toupper`, `trim`, `trimleft`, `trimright`.
`switch`	Multi-way branch.
`tell`	Return the current seek offset of an I/O stream.
`time`	Measure the execution time of a command.
`trace`	Monitor variable assignments.
`unknown`	Unknown command handler.
`unset`	Undefine variables.
`uplevel`	Execute a command in a different scope.
`upvar`	Reference a variable in a different scope.
`while`	A loop construct.

CHAPTER 2

Tcl Data Structures

This chapter describes the two main 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 2–1 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 2–2 Using `info` to determine if a variable exists.

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

In Chapter 4 there is an example on page 45 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 dynamically.

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 2–1 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 2–3 Constructing a list with the `list` command. Syntax may be added to the result by `list` so each argument becomes one list element.

```
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 2–4 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 2–5 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 2–6 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 5.

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.

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

The `lrange` command returns a range of list elements. It takes a list and two indices as arguments. The keyword `end` may be used to specify the last element of the list. (Unfortunately, `end` does not work with `lindex`).

```
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. (`linsert` does not support the `end` keyword.) 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. As with `lrange`, the index of the last list element can be specified with the `end` keyword.

Example 2–7 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
```

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 6. 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 2–8 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 2–9 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.

SPLIT AND JOIN

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 2–10 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 2–11 Using arrays to compute the product of 1 through 10.

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

This example sets `arr(x)` to the product of `1 * 2 * ... * x`. The initial assignment of `arr(0)` defines `arr` as an array variable. The example uses the variables `j` and `product` just for clarity. The loop body could have been the single statement:

```
set arr($i) [expr $i * $arr([expr $i-1])]
```

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 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.

Example 2–12 What if the name of the array is in a variable.

```
set name TheArray
set ${name}(xyz) {some value}
=> can't set "name": variable is array.
set [set name](xyz) {some value)
=> some value
```

The Array Command

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

Table 2–2 The array command

`array names` *`arr`*	Return the list of indexes defined for *`arr`*.
`array size` *`arr`*	Return the number of indexes 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.

Table 2–2 The array command

`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 34.)

```
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.

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.

A note about variable scope

The next two sections discuss arrays in the context of variable scope, which is considered again in more detail in Chapter 4. A variable's *scope* is the range of the program over which the variable is defined. In Tcl, each procedure defines a new scope for its variables, and the variables become undefined when the procedure returns. There is also a *global scope* for variables defined outside of any procedure. Global variables are *not* visible inside procedures until they are declared with the `global` command. The `upvar` command is used to reference variables in other procedure scopes. Again, if this does not make sense yet, skip to Chapter 4 and come back to this section later.

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` (or `upvar`) 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 OBJECTS.

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 2–13

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.

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 2–14 `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]
```

* 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.

```
        }
    }
    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
```

Passing Arrays By Name

There are two ways to gain access to an array inside a procedure. The first is by using the `global` command as in the previous two examples. It is also possible to 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.

Example 2–15 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.

CHAPTER 3

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 3–1 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 3–2 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 6. 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 57.
`-regexp`	Use regular expression pattern matching. See page 58.
`--`	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 3–3 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 3–4 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 58.)

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 3–5 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 3–6 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 3–7 Parsing command line arguments. `argv` is initialized by the Tcl interpreter.

```
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 3–8 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 3–9 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 3–10 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 T cl.

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 3–11 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 3–12 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 3–13 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 3–14 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 3–15 Specifying errorinfo with return.

```
proc bar {} {
    return -errorinfo "I'm giving up" bogus
}
bar
=> 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 4

Procedures and Scope

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

*P*rocedures 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 4–1 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 4–2 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
```

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

* 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.

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 4–3 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 commands 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 4–4 A random number generator. (Adapted from “Numerical Recipes in C” by Press et al. Cambridge University Press, 1988)

```
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
```

Call By Name - 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 2–15 on page 30.)

Example 4–5 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 4–6 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
}
```

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]
```

Info Level

The `info` command returns information about the state of the interpreter. The info level option returns information about the Tcl call stack. This can be useful to debuggers or other code that needs to search the stack. The syntax is:

```
info level ?number?
```

With no *number* argument, this command returns the current level of the Tcl call stack. If a number is specified, then the name and arguments of the command at that level are returned in a list. The following code provides a simple call trace.

Example 4–7 Getting a trace of the Tcl call stack.

```
proc CallTrace {} {
    for {set l [info level]} {$l > 0} {incr l -1} {
        puts stdout [info level $l]
    }
}
```

CHAPTER 5

Eval

This chapter concerns the execution of dynamically generated code using the `eval` command.

One of the troublesome spots in Tcl is the evaluation of dynamically generated code. This can crop up in simple cases, such as dealing with the list-valued `args` parameter to a procedure. It also arises with the Tk `send` command in which commands are sent to remote interpreters.

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, too, 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. W e 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 5–1 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 5–2 `PackedButton` creates and packs 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

There are two Tk commands, `after` and `send`, that also `concat` 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.

CHAPTER 6

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 6–1 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 57
`string range` *`str first last`*	Return the range of characters in *`str`* from *`first`* to *`last`*.
`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.

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 6–1 Comparing strings.

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

APPEND

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.

FORMAT

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.

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

Table 6–2 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 .

`$` 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 6–3 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
=> Label                2
```

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
=> Key      Value
```

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.)

STRING MATCH

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
```

* 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.

```
=> 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
```

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 6–4 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.

Table 6–4 Regular Expression Syntax

^	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 6–2 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*

As with `regexp`, `regsub` returns 0 or 1 to indicate whether or not the string matched the pattern. In addition, `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 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 command syntax for pipelines and 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 output.
`>> 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 The `newer` procedure to compare file 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, creating intermediate directories as needed.

```
proc makedir { pathname } {
    if {[file isdirectory $pathname]} {
        return $pathname
    } elseif {[file exists $pathname]} {
        error "Non-directory $pathname already exists."
    } else {
        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. Optional arguments are delimited with ?.

`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 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 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.

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`.

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 .

MISCELLANY - EXIT AND PID

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

Tk Fundamentals

This chapter introduces the basic concepts used in the Tk toolkit for the X window system.

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 eventually. 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 term *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 10.

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 11. 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` interpreter, or you can provide the event loop in your own C program as described in Chapter 14.

Event binding is done on a per-widget basis. Unlike some other toolkits, input events are not passed up through the widget hierarchy. Instead, each widget must decide what events it will handle, and if a particular event is not handled it is discarded. (Tk 4.0 supports class and global bindings better than earlier versions of the toolkit.)

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.

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.

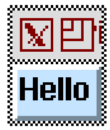
If you use `wish` interactively, it will create and display an empty main window and give you a command line prompt. With this interface, your keyboard commands are handled by the event loop, so you can build up your Tk interface gradually. As we will see, you will be able to change virtually all aspects of your application interactively.

HELLO WORLD IN TK

Example 8–1 "Hello, World!" Tk program.

```
#!/usr/local/bin/wish -f
button .hello -text Hello \
   -command {puts stdout "Hello, World!"}
pack .hello
```

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.

```
pack .hello
```

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 that is packed inside it. The behavior of the packer will be discussed further in Chapter 9 and Chapter10Tk 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 T cl 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
```

Without a value argument, the config option returns the current attribute settings. Unfortunately, this includes the command line switch, the resource name, the class name, the default value, and the current value, which is last. You have to use the lindex command to extract the current value:*

```
set oldbg [lindex [.hello config -background] 4]
```

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
```

* Tk 4.0 widgets will provide a more direct way to get the current value.

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 command 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 only have to worry about these naming conventions if you specify widget attributes via the X resource mechanism, which is described in more detail in Chapter 12.

Summary Of The Tk Commands

The following two tables list the Tcl commands added by Tk. The first table lists commands that create widgets. The second table lists commands that manipulate widgets. There is no chapter devoted to the basic widgets. However, nearly all the Tk commands are used in one example or another. Look through the index to find out where they are used in the book.

Table 8–1 Tk widget-creation commands

`button`	Create a command button.
`checkbutton`	Create a toggle button that is linked to a Tcl variable.
`radiobutton`	Create one of a set of radio buttons that are linked to one variable.
`menubutton`	Create a button that posts a menu.
`menu`	Create a menu.
`canvas`	Create a canvas, which supports lines, boxes, bitmaps, arcs, text, polygons, and embedded widgets.
`label`	Create a read-only, one-line text label.
`entry`	Create a one-line text entry widget.
`message`	Create a read-only, multi-line text message.

Table 8–1 Tk widget-creation commands

`listbox`	Create a line-oriented, scrolling text widget.
`text`	Create a general purpose text widget.
`scrollbar`	Create a scrollbar that can be linked to another widget.
`scale`	Create a scale widget that adjusts the value of a variable.
`frame`	Create a container widget that is used with geometry managers.
`toplevel`	Create a frame that is a new top level X window.

Table 8–2 Tk widget-manipulation commands

`after`	Execute a command after a period of time elapses.
`bind`	Bind a Tcl command to an X event.
`destroy`	Delete a widget.
`focus`	Control the input focus.
`grab`	Steal the input focus from other widgets.
`lower`	Lower a window in the stacking order.
`option`	Access the Xresources database.
`pack`	The packer geometry manager.
`place`	The placer geometry manager.
`raise`	Raise a window in the stacking order.
`selection`	Manipulate the X PRIMARY selection.
`send`	Send a Tcl command to another Tk application.
`tk`	Query internal Tk state (e.g., the color model)
`tkerror`	Handler for background errors.
`tkwait`	Block awaiting an event.
`update`	Update the display by going through the event loop.
`winfo`	Query window state.
`wm`	Interact with the window manager.

C H A P T E R 9

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 10 and event bindings are discussed in Chapter 11.

Pingtool, Version 1

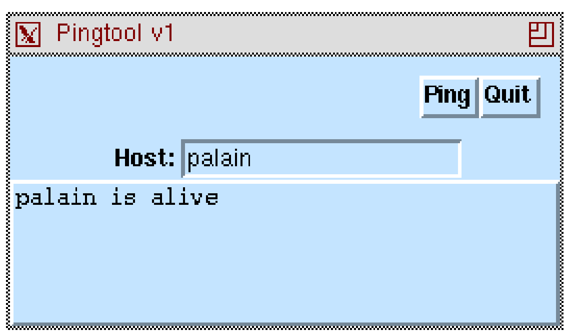
Our first example provides a simple user interface to the `ping` program that is used to query the network connectivity of hosts. The interface will consist of two buttons, `Ping` and `Quit`, an `entry` widget in which to enter a hostname, and a `text` widget in which to log the results of `ping`.

Example 9–1 Pingtool, Version #1

```
#!/usr/local/bin/wish -f
# pingtool - a simple user interface to ping

# Set title and allow window resizing.
wm title . "Pingtool v1"
wm minsize . 20 20
```

```
# Create a frame for buttons.
frame .buttons -borderwidth 10
pack .buttons -side top -fill x

# Create the command buttons.
button .buttons.quit -text Quit -command exit
button .buttons.ping -text Ping -command Ping
pack .buttons.quit .buttons.ping -side right

# Create a labeled entry for hostname type in.
frame .f ; pack .f -side top
label .f.l -text Host:
entry .f.host -width 20 -relief sunken
pack .f.l .f.host -side left

# A Return triggers the Ping command
bind .f.host <Return> Ping

# Create a text widget to log the output
text .log -width 40 -height 5 -bd 2 -relief raised
pack .log -side top

# Run ping and log the results
proc Ping {} {
    set hostname [.f.host get]
    catch {exec -keepnewline ping -v $hostname} result
    .log insert end $result
    .log yview -pickplace end
}
```

The user interface is show below . The title bar at the very top of the window is provided by the window manager , which is `twm` in this case.



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 . "Pingtool v1"
```

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.)

The next command sets the minimum size for the window. This has the important side effect of allowing the window to be resized. Without this command (or a `wm maxsize` command), the size of the main window cannot be changed interactively.

A frame is created to hold the buttons. It has a border to provide some space around the buttons:

```
frame .buttons -borderwidth 10
```

The button frame is positioned in the main window.

```
pack .buttons -side top -fill x
```

The frame will not be visible until it is packed. 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.

Two buttons are created: one to ping the host, the other to quit the program. Their names, `.buttons.quit` and `.buttons.ping`, imply that they are children of the `.buttons` frame.

```
button .buttons.quit -text Quit -command exit
button .buttons.ping -text Ping -command Ping
```

The default location for a widget is inside its parent frame. The buttons are arranged inside their parent frame on its right side.

```
pack .buttons.quit .buttons.ping -side right
```

The frame for the `label` and `entry` is created and positioned in the main window.

```
frame .f ; pack .f -side top
```

Note the lack of border for `.f` and the lack of the `-fill x` packing option. The missing fill causes the `.f` frame to shrink down to be just big enough to hold its children widgets. It might help you to specify different background colors (e.g., `-background red`) for the different frames in your scripts in order to visualize the effects of the different packing parameters.

The `label` and `entry` are created as children of this frame. The size of the `entry` is specified in terms of characters. The `-relief sunken` attribute gives the `entry` some looks to set it apart visually on the display.

```
label .f.l -text Host:
entry .f.host -width 20 -relief sunken
```

These are packed side-by-side in their parent frame.

```
pack .f.l .f.host -side left
```

A binding is created for the entry widget that makes it run `ping` when the `Return` key is pressed.

```
bind .f.host <Return> Ping
```

The `bind` command associates a Tcl command with an X event in a particular widget. In this example we are binding a command to the `<Return>` event, which is the event generated when the user presses the Return key on the keyboard. The command is just Ping, which is the same command that is bound to the Ping button. For the event to go to the entry widget, .f.host, 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. You'll need to do that before you can type anything into the widget.

Finally, a text widget, .log, is created and packed into the bottom of the window.

```
text .log -width 40 -height 5 -bd 2 -relief raised
pack .log -side top
```

Here we use the -bd abbreviation for -borderwidth. The width and height of the text widget are specified in terms of characters. Note that there is no scrollbar. That would be a separate widget, and we'll show how to add one on page 90. In the meantime, the default bindings for a text widget allow you to scroll it by pressing and dragging with the middle mouse button.

The heart of the script is the Ping procedure. The first line of Ping uses the name of the entry widget as a command in order to retrieve its current value.

```
set hostname [.f.host get]
```

This illustrates a common property of all Tk widgets. When you create a widget, a new Tcl command is created with the same name as the widget.The new command operates on that instance of the widget. In this case, the command .f.host get returns the text currently in the widget.

The second line executes the ping program (finally!).

```
catch {exec -keepnewline ping -v $hostname} result
```

The exec command is wrapped up with catch to guard against possible errors. For example, the host name could be bogus. In any case, the output of the ping program gets stored into the result variable. The newline is retained in the exec results so the result appears on one line in the text widget.

The result is inserted into the log:.

```
.log insert end $result
```

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 contents of the text widget are positioned in the window so the newly inserted line is visible.

```
.log yview -pickplace end
```

The text widget's yview function scrolls the widget so that a particular mark is visible. The -pickplace option gives the widget some flexibility as to where that mark is visible in the widget. Otherwise the given mark is displayed at the top line of the widget.

To summarize, either a button click on the Quit button or a <Return> in the entry widget will invoke the Ping procedure. That retrieves the current value from the entry, runs the ping program, and displays the result.

PINGTOOL, VERSION 2

The main problem with the first version of pingtool is that it cannot be used to run `ping` in its continuous mode where it periodically sends a packet and reports the time taken to receive a response. The problem is that the `exec` of ping -s would not terminate until `ping` was interrupted with a signal, so the user interface would not get any output from ping.

We'll address this problem by introducing a helper script and using the Tk `send` command to communicate between them. The `send` command lets one Tcl interpreter invoke a Tcl command in another interpreter, which is a very general and powerful mechanism. The use of `send` establishes a relationship that is a bit different than a pipeline between two processes. With a pipeline, the receiver is normally dedicated to reading the input. With Tk `send`, the receiver can process all types of X events, such as button hits and keystrokes, along with the messages used to implement `send`. In addition, the receiver could get commands from a number of sources, not just a single one. Thus, by using a background process that sends Tcl commands to the user interface, the user interface can remain active while the background process runs.

The user interface script will remain much the same as the original version. The `Ping` procedure will be modified to `exec` the helper script instead of running the `ping` program directly. A `Stop` procedure will be added that kills the helper script. Finally, an `Insert` procedure will be added for the convenience of the helper script. The helper script will be run in the background, and it will send `Insert` commands to the user interface in order to display lines of output.

Example 9–2 Pingtool, Version #2

```
#!/usr/local/bin/wish -f
# pingtool - supports continuous ping via a helper script
# Set title and allow resizing.
wm title . "Pingtool v2"
wm minsize . 20 2

# Create a frame for the buttons
frame .buttons -borderwidth 10n
pack .buttons -side top -fill x

# Create the command buttons
button .buttons.quit -text Quit -command exit
button .buttons.ping -text Ping -command Ping
pack .buttons.quit .buttons.ping -side right

# Create a labeled entry for the hostname type in.
frame .f ; pack .f -side top
label .f.l -text Host:
entry .f.host -width 20 -relief sunken -textvariable host
pack .f.l .f.host -side left

# Pressing return in the entry triggers the ping.
bind .f.host <Return> Ping
```

```
# Create a text widget to log the output
text .log -width 60 -height 5 -bd 2 -relief raised \
   -setgrid true
pack .log -side top

# Run the helper script in the background
# and change the Ping button to a Stop button.
proc Ping {} {
   global host pid
   set pid [exec pinghelper $host [winfo name .] &]
   .buttons.ping config -text Stop -command Stop
}
# Insert stuff into the log, appending a newline.
proc Insert { s } {
   .log insert end $s
   .log insert end \n
   .log yview -pickplace end
}
# Stop the background helper process.
proc Stop {} {
   global pid
   catch {exec kill $pid}
   .buttons.ping config -text Ping -command Ping
}
```

The `entry` widget uses the `-textvariable` option in this version.

```
entry .f.host -width 20 -relief sunken -textvariable host
```

The `-textvariable` option is being used to tie the global Tcl variable `host` to the value being displayed in `.f.host`. The current value stored in the widget is also the value of the Tcl variable, and changes to the variable value change what is displayed. This makes it a little easier to get at the current value.

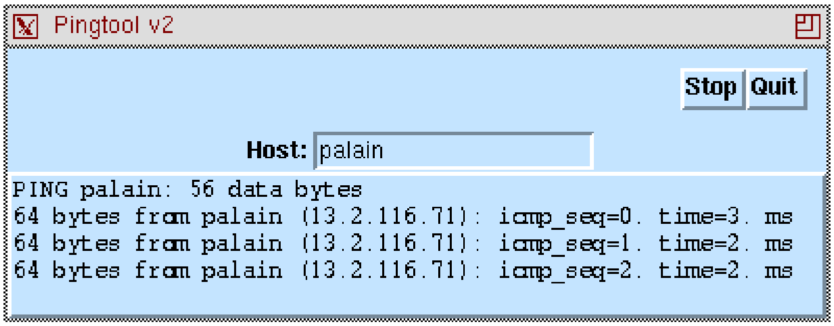
The variable `host` is visible in the global scope, so the `Ping` procedure must use the `global` command to make it visible inside the procedure. Similarly, the `pid` variable is used to communicate the process id of the helper script between `Ping` and `Stop`, so it is shared via the global scope.

The `-setgrid true` option is used on the text widget. This changes the meaning of the width and height parameters in the `wm minsize` command from pixels to character units. When the window is resized, the geometry reported by the window manager will be in character units.

Instead of adding another button for the `Stop` command, this script changes one button back and forth between a `Ping` and a `Stop` button.

```
.buttons.ping config -text Stop -command Stop
.buttons.ping config -text Ping -command Ping
```

This illustrates the ability to reconfigure a Tk widget dynamically. Any attributes that you can specify when you create a widget can be changed later by using the widget's `configure` function, which can be abbreviated to `config`.



Example 9–3 Ping helper script that uses send.

```
#!/usr/local/bin/wish -f
# pingtool helper script

# Unmap the unneeded window.
wm withdraw .

# Get command line arguments.
set hostname [lindex $argv 0]
set otherInterp [lindex $argv 1]

# Start the ping pipeline.
set in [open "|ping -s $hostname" r]

# Read ping results and send results to the front end.
while {[gets $in line] >= 0} {
    send $otherInterp [list Insert $line]
}
exit
```

The helper script does not need a window, so it's first command is a window manager command to unmap its window from the display.*

```
wm withdraw .
```

The next two lines get the command line arguments. The `argv` variable is a list of the command line arguments. It does not include the command name itself, so the zeroth element is the first argument. `argc` counts the arguments (it is not used here), and it would be equal to 2 in this case.

```
set hostname [lindex $argv 0]
set otherInterp [lindex $argv 1]
```

The value of `otherInterp` is a name used in the `send` command to identify which Tcl interpreter should receive the command. This value is obtained by the

* However, it is still necessary to use `wish`, and not `tclsh` for example, because the `send` command is implemented using window system facilities

user interface with the `wm name .` command and passed to the helper script as an argument.

```
set pid [exec pinghelper $host [winfo name .] &]
```

The `wish` shell derives the name of the interpreter from the filename of the script, or from the `-n` command line argument. However, in the case of multiple instances of a program running on the same display, a unique number is appended to this name. So, you cannot always predict the name of an application. Passing the results of the `wm name .` command is better than putting the name directly into scripts.

The helper script uses `open` to create a pipeline between it and the `ping` program. This causes the `ping` program to be forked and run in the background, and its output to be available to the script via an I/O stream.

```
set in [open "|ping -s $hostname" r]
```

The script then goes into a read loop. The `while` command will execute until the `ping` program stops producing output. The `gets` command places one line of input into the `line` variable, and discards the trailing newline. It returns the number of characters read, not counting the newline, or -1 upon end of file.

```
while {[gets $in line] >= 0} {
```

The inner loop uses the `send` command to communicate with the user interface.

```
send $otherInterp [list Insert $line]
```

The command sent is the result of `list Insert $line`. This ensures that `Insert` will be called with a single argument in the remote interpreter. Recall that `list` constructs a list with one element for each of its arguments, quoting things if necessary. By using `list`, we insulate the script from any funny effects that special characters in the input could have on the remote Tcl parser.

To emphasize this point further, consider the following incorrect alternatives:

```
send $otherInterp {Insert $line}
send $otherInterp "Insert $line"
send $otherInterp Insert $line
```

The first alternative is clearly wrong because `$line` will not be substituted at all, resulting in a bunch of "`$line`" strings being inserted into the log. The second alternative will only work if the input line contains a single Tcl list element. Otherwise the remote interpreter will complain that too many arguments are being passed to `Insert`. The reason this happens is that the remote interpreter completely reparses the whole string as a Tcl command. Because the substitution of `$line` occurs in the helper script's interpreter, by the time the result is seen by the user interface interpreter there is no longer any structure to the string. The use of the `list` command ensures that the structure is maintained. The third alternative is essentially the same as the second because `send` uses `concat` to combine any extra arguments into a single command string.

ANIMATION WITH UPDATE

Another way to address the problem with the first version of pingtool is to animate the user interface explicitly by using the `update` command that causes Tk to go through its event loop and process any pending events. In this case we will move the read loop of the `ping -s` pipeline back into the user interface script. The loop will contain an `update` call to keep the interface active. This technique has some potential problems, however, as discussed below.

Example 9–4 Pingtool that uses the update command.

```
#!/usr/local/bin/wish -f
# pingtool - a simple user interface to ping
wm title . "Pingtool v3"
wm minsize . 10 10

frame .buttons -borderwidth 10
pack .buttons -side top -fill x

button .buttons.quit -text Quit -command exit
button .buttons.ping -text Ping -command Ping
pack .buttons.quit .buttons.ping -side right

frame .f ; pack .f -side top
label .f.l -text Host:
entry .f.host -width 20 -relief sunken -textvariable host
pack .f.l .f.host -side left

bind .f.host <Return> Ping

text .log -width 60 -height 10 -bd 2 -relief raised \
    -setgrid true
pack .log -side top

proc Ping {} {
    global host stop
    set input [open "|ping -s $host" r]
    set stop 0
    # The stop button sets the global stop variable.
    .buttons.ping configure -text Stop -command {set stop 1}
    while {[gets $input line] >= 0} {
        Insert $line
        # Update forces screen update and allows handling
        # of button hits and keystrokes.
        update
        if {$stop} {
            break
        }
    }
    # Clean up the pipeline and restore the Ping button.
    catch {close $input}
    .buttons.ping configure -text Ping -command Ping
}
```

```
proc Insert { s } {
    .log insert end $s\n
    .log yview -pickplace end
}
```

As a side effect of the `update` command, a button press on the `Stop` button will set the `stop` variable. The `Ping` procedure must explicitly poll the `stop` variable to detect if it is time to break out of the loop. The button action is executed at the global scope, so the `stop` variable must be declared `global` so that `Ping` and the button command access the same variable.

The background ping process is stopped by closing the pipeline I/O stream, which will result in a `SIGPIPE` to the `ping` process the next time it writes to its standard output. If the pipeline was closed as a direct response to the `Stop` action, then the read loop would have to deal with the possibilities of errors from `gets`. Finally, as a matter of habit I wrap up the pipeline `close` with a catch phrase. If there were any errors raised by the pipeline they appear at the time of the `close`. This includes writing to standard error, as well as the processes exiting with a non-zero status code.

The Danger of update

There is one flaw in this script due to its use of `update`. Consider the `bind` command that calls `Ping` if you press `<Return>` in the entry widget. If you run this example you will find out that it is possible to start a second `ping` process and associated read loop by changing the hostname and pressing `<Return>` while one `ping` process is already active. What happens is that `Ping` is activated again, essentially inside the `update` command. The original `Ping` command is suspended until its `update` command completes. However, when the `update` command processes the `<Return>` event it calls `Ping` again, and `Ping` will run until the `Stop` button is clicked, or yet another `ping` process is started in the same way.

For this reason I avoid the use of `update`. In more complex applications it can result in unexpected behavior that is difficult to understand. There is, however, a variation on `update` that only causes the display to be updated but does not process any input events. This is:

```
update idletasks
```

The Tk widgets do their display operations in an idle-time callback, so this command causes any pending display operations to be performed. By doing things at idle time, the widgets can batch up several display changes and do them all at once. By explicitly calling `update idletasks`, you force the widgets to redisplay themselves without the danger of handling input events that might spawn arbitrary Tcl commands.

ADDING SCROLLBARS

The Tk widgets do not have built in scrollbars. Instead, a `scrollbar` is a separate widget that you must couple to another widget explicitly. Other widgets that often have scrollbars associated with them include the `text`, `canvas`, and `listbox` widgets. The scrollbar and the other widget are associated via Tcl commands. When the scrollbar is adjusted, it executes a Tcl command that is registered with the scrollbar that typically makes the other widget change its display. Similarly, when the widget changes its display, it executes a Tcl command that typically causes the scrollbar to update its own display. Thus the two are kept in sync.

Example 9–5 A scrollbar and a text widget.

```
proc ScrolledText { f args } {
    # Create a frame to hold text and scrollbar
    frame $f

    # Create the text widget, passing through extra
    # arguments using eval and args. (See also page 49.)
    eval {text $f.text -yscroll [list $f.scroll set]} $args

    # Create the scrollbar, hooking it to the text widget.
    scrollbar $f.scroll -orient vertical \
        -command [list $f.text yview]

    # Pack the scrollbar first so it cannot get clipped.
    pack $f.scroll -side left -fill y
    pack $f.text -side left -fill both -expand true
    pack $f -fill both -expand true
    return $f.text
}
```

The `eval` command is used to construct and evaluate the `text` command. This is necessary because the list-valued parameter `args` may contain additional parameters for the text command. The internal `concat` done by `exec` will join these arguments to the rest of the `text` command. The bulk of the `text` command is wrapped up in braces to it only gets evaluated one time.

The `-yscrollcommand` (abbreviated `-yscroll` here) attribute of the text widget defines part of the Tcl command executed when the `text` widget changes its display. Additional parameters are appended to this command based on the size of the window and the amount of text displayed in it relative to the total amount of text in the widget. The value for `-yscroll` is formed with list so that `$f` gets substituted in the current context. When the scroll command is evaluated later, the parameter `f` will no longer be defined.

Similarly, the `-command` parameter for the `scrollbar` defines part of the Tcl command executed when the `scrollbar` is manipulated. An additional parameter is appended to this by the `scrollbar` implementation to get a particular line to appear at the top of the widget display. This command is formed using list so that `$f` gets substituted in the right context.

It is important to pack the `scrollbar` into the frame first, otherwise it could get clipped out of the display when the window is resized. Its pack parameters include a `-fill y` so that it fills out the whole height of the frame as it changes size. The text widget is packed with `-fill both -expand true` so that it will expand as the frame expands, and fill up that expanded area with its display. (The distinction between fill and expand is discussed in Chapter 10.)

(Consider adding simple dialog box example here.)

CHAPTER 10

The Pack Geometry Manager

This chapter explores the `pack` geometry manager that is used to position widgets on the screen.

The packer is a powerful geometry manager used to position windows. 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 with a few simple 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.

Example 10–1 A simple application template used to demonstrate the packer.

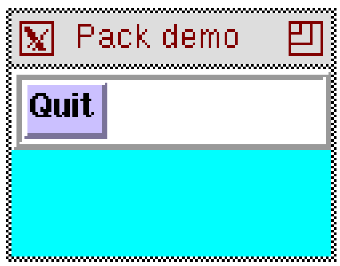
```
proc SimpleTkApp1 { name } {
    . config -bg black
    frame .buttons -borderwidth 4 -relief ridge -bg white
    button .buttons.quit -text Quit -command exit -bg pink
    frame .body -width 120 -height 40 -bg yellow
    pack .buttons -side top -fill x
    pack .body -side top
    pack .buttons.quit -side left
    wm title . $name
    wm minsize . 30 30
```

```
}
SimpleTkApp1 "Pack demo"
```

The `SimpleTkApp1` procedure provides a basic skeleton that you can reuse for many simple interfaces. It initializes the widget tree to have row of buttons along the top and a main body area underneath that. By defining a procedure like `SimpleTkApp1` in your Tcl library, you can save a little typing when creating new tools, and you ensure a common structure to your user interfaces. Of course, this template is very simple, and in practice you will add more features to this basic template. Additional items might include an entry widget used for status messages, and a help button to explain the tool to users.

Horizontal And Vertical Stacking

The initial widget framework from `SimpleTkApp1` is show below. (The title bar along the top is implemented by the window manager.)



At the top level of the widget tree, two frames are created and packed into a vertical stack.

```
frame .buttons -borderwidth 4 -relief ridge -bg white
frame .body -width 120 -height 40 -bg yellow
pack .buttons -side top -fill x
pack .body -side top
```

The `-side top` arguments to `pack` position the frames towards the top of the window, and the first one packed is uppermost.* The width of the vertical stack is determined by the widest member, which in this case is `.body` with its width specified to be 120 pixels.

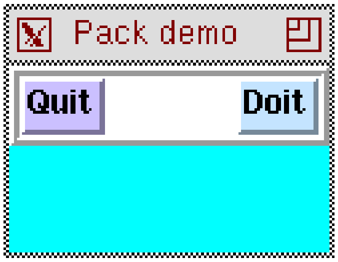
A frame with nothing in it will keep its specified size, like `.body` does here. Once a widget is packed inside a frame, however, it shrinks down to be just big enough to hold its children. No size is specified for `.buttons`. Instead the `-fill x` packing parameter causes the `.buttons` frame to occupy the whole top side of the main window.

Within the button frame, `.buttons`, the `Quit` button is packed to the left.

```
pack .buttons.quit -side left
```

* This is similar in spirit to a Vbox or VerticalPane in other window toolkits.

More buttons can be created and packed into a horizontal stack by specifying the `-side right` or `-side left` pack option.

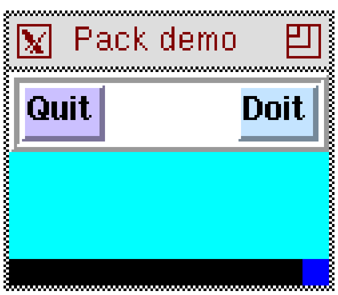



```
button .buttons.doit -text Doit
pack .buttons.doit -side right
```

In general, 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`. This is not strictly necessary, but it leads to predictable behavior by the packing algorithm.

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. After each widget is packed, the cavity changes. The main trick is that a widget occupies one whole side of the cavity, even if there appears to be more room along that side. In our example so far, both `.buttons` and `.body` have been packed towards the top side. Thus the cavity is below them, although it is not visible.



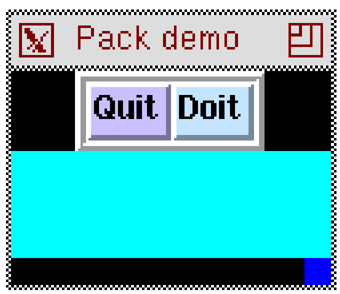

```
frame .little -width 10 -height 10 -bg red
pack .little -side right
```

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 below the two frames already packed toward the top side. The frame does not appear to the right of the existing frames as you might have expected. After `.little` is packed, the black background of the main frame is visible where the cavity is unfilled. As a general rule, mixing horizontal and vertical stacking inside the same frame can lead to unexpected behavior. It is confusing because the notion of the cavity is maintained internally by the packer and it is often not visible. The best way to deal with this is confine yourself to vertical or horizontal stacks within a frame.

Packing Space Vs 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.

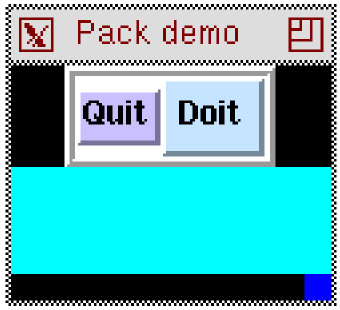
The `-fill` option causes a widget to fill up the allocated packing space with its display. For example, the `-fill x` packing option for `.buttons` causes it to fill up the display in the horizontal direction.




```
pack .buttons -fill none
```

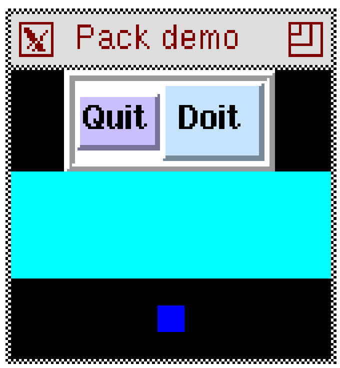
The `pack` command lets you change packing parameters dynamically. If we turn the fill off, then the `.buttons` frame shrinks down to be just big enough to hold its widgets, taking into account its own border width. The main frame shows through on either side of `.buttons` because there is unfilled packing space there. Similarly, there are no fill options for `.little`, so the cavity at the bottom remains unfilled, too.

The `-ipadx` and `-ipady` packing options give a widget more display space in the x and y directions. The following makes the `Doit` button bigger:




```
pack .buttons.doit -ipadx 4 -ipady 4
```

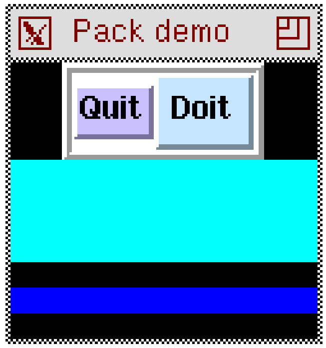
In contrast, the `-padx` and `-pady` packing options allocate more packing space, but not display space.




```
pack .little -side top -pady 10
```

This last change has changed the packing side for .little, which centers it against the top of the cavity. Extra padding space has been allocated in the y direction. However, because no fill is specified, the parent frame shows through from behind.

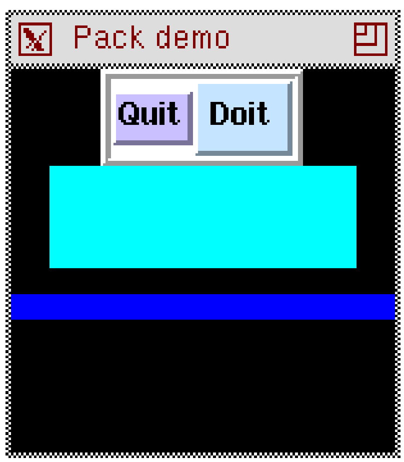
The y padding is more evident when we let `.little` fill in the x direction.:




```
pack .buttons .little -fill x
```

Expand And Resizing

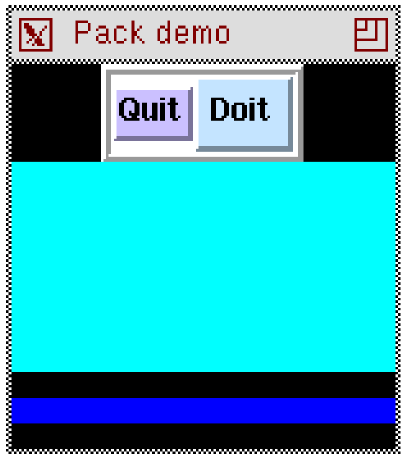
Now we can investigate what happens when the window is made larger. A side-effect of the `wm minsize` command in `SimpleTkApp1` is that it allows the main window to be resized by the user. We can also specify the size with the `wm geometry` command. When the window is resized either way, the packer has to decide who gets the extra space.



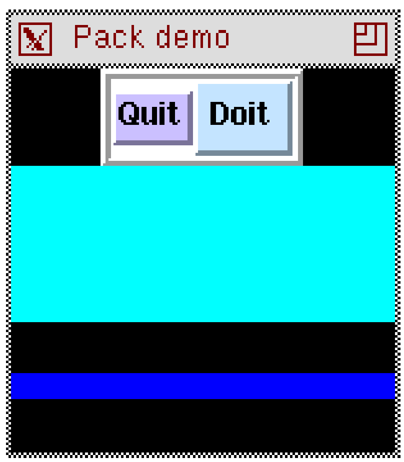

```
wm geometry . 150x150
```

The only widget that claims any of the new space is `.little` because of its `-fill x` packing option.

The `-expand` option tells the packer to allocate extra space to that widget. Typically the main frame is set up to expand this way .




```
pack .body -expand true -fill both
```

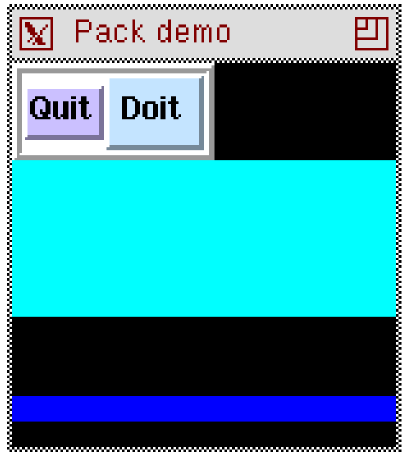



```
pack .little -expand true
```

If more than one widget at the same level is allowed to expand, then the packer shares the extra space between them. In the bottom figure, both `.body` and `.little` are competing for the space. The `-fill both` packing parameter for `.body` causes it to fill its extra packing space with its display . The `.little` frame only has fill turned on in the x direction, so it has unfilled packing space.

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`.

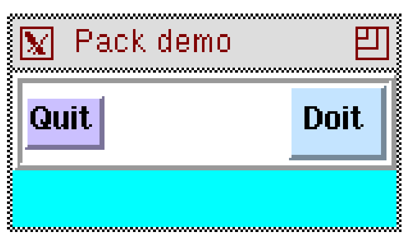



```
pack .little -anchor s
pack .buttons -anchor nw
```

The `nw` anchor for `.buttons` has the same effect as `w` or `sw` because there is no free space above or below `.buttons`. The y padding for `.little` keeps it from touching the bottom of the main frame even though it has `-anchor s`.

Packing Order

The last lesson to learn about the packer is the effect of packing order. In our case `.buttons` was packed first, then `.body`, and last we packed `.little`. 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.



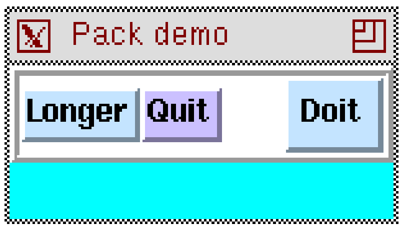

```
pack .buttons -fill x
wm geometry . 150x60
```

The `.little` frame has been clipped because it was last in the packing order. By default a newly packed widget goes last in the packing order. The

`-after` and `-before` options give you control over the packing order. Recall that the buttons have been packed with these commands, which place `.buttons.quit` at the front of the packing order.

```
pack .buttons.quit -side left
pack .buttons.doit -side right
```

Packing another button to either the left or right side would position the new button between the existing buttons because that is where the cavity is. However, if we pack the new button in at the front of the packing order, then the placement is recomputed from scratch.




```
button .buttons.long -text Longer
pack .buttons.long -before .buttons.quit -side left
```

Equal-Sized Buttons

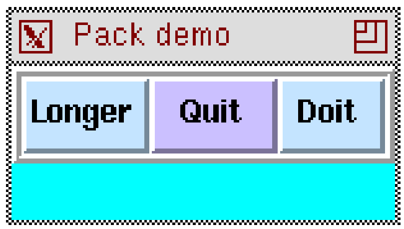
A button's size is determined by the text or graphic that is displayed on it, as well as the internal padding, expand, and fill options. In addition, its width can be specified, in characters, in order to get buttons of a fixed size. There are at least two reasons why you might want to do this. The first is that you just want equal-sized buttons. The second is if you will be changing the button text dynamically. If you set a fixed size that is large enough for all labels, then the display will not repack when you change the button text.

Example 10–2 Making equal sized buttons.

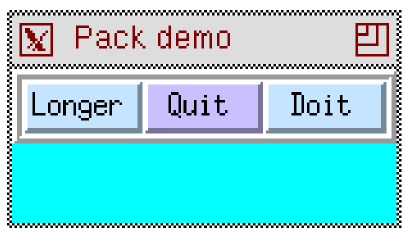
```
proc SetButtonSizes { frame {min 0} } {
    set maxl $min

    # Query each widget in the frame for its width.
    foreach w [pack slaves $frame] {
        set len [string length \
            [lindex [$w config -text] 4]]
        if {$len > $maxl} {
            set maxl $len
        }
    }
    # Set all the widths to be the same.
    foreach w [pack slaves $frame] {
        $w config -width $maxl
```

```
    }
    # Repack the widgets so they expand into any extra
    # space and fill that space with their display.
    # eval is used because pack slaves returns a list.
    eval pack [pack slaves $frame] -expand true -fill both
}
SetButtonSizes .buttons
```

The button sizes are set in two steps. First they are all made the same width. Then they are told to expand and fill the extra space. The `pack slaves` command is used to find out what widgets are packed inside the frame.



If you look closely, the `Doit` button is still a bit smaller than the other buttons. This is due to the variable width fonts that are used and the fact that button widths are specified in characters, not pixels. If the widget sizes are different before they expand, they will be different afterwards, too. If the buttons are configured with a fixed-width font, they get the same size.




```
pack .buttons.doit -ipady 0 -ipadx 0
foreach b {long quit doit} {
    config .buttons.$b -font fixed
}
```

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.
- 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, and then applies the other packing options to that base size.
- 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 that is otherwise unclaimed. If more than one widget at the same level wants to expand, then they share the extra space.
- The `-ipadx` and `-ipady` options allocate more display space (if possible).
- The `-padx` and `-pady` options allocate more packing space (if possible).

C H A P T E R 11

Binding Commands to X Events

This chapter introduces the event binding mechanism in Tk. The extended example is a user interface to view and edit bindings.

Binding associates 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 key sequences for which there are command bindings.

```
bind Listbox
=> .<B2-Motion> <Button-2> <Shift-B1-Motion>
    <Shift-Button-1> <B1-Motion> <Button-1>
```

These events are button-related events. `<Button-2>` for example, is the event generated when the user presses the second, or middle, mouse button. `<B1-Motion>` is a mouse motion event modified by the first, or left-hand, mouse button. This event is generated when the user drags the mouse with the left button pressed. The sequence syntax will be described in more detail below.

If `bind` gets a key sequence argument then it returns the Tcl command bound to that sequence:

```
bind Listbox <Button-1>
=> %W select from [%W nearest %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 `%y` keyword is replaced with the widget-relative Y coordinate of the event. 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 command can be registered for a widget, a class of widgets, or all widgets. The keyword `all` means the binding applies to all widgets. There is a precedence among these sources of bindings, and 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.

The default behavior for Tk widgets is defined by a set of class-specific bindings. In Tk 3.6 and earlier, 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 ensures that the class-specific binding is executed before the new binding.

```
bind .list <Button-1> "[bind Listbox <Button-1>] ; Doit"
```

The binding precedence mechanism will change for the better in Tk 4.0. Each source of bindings (`all`, class, and instance) can register a command for an event. Override is achieved with an explicit `break` in a binding command body.

Events

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`. The surrounding brackets are used to delimit a single event. If there are no brackets, then the characters are treated as literals in a sequence, all of which must match.

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 11–1 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.
`CirculateRequest`	Generated before a change of stacking order.
`Colormap`	The colormap has changed.
`Configure`	The window has changed size, position, border, or stacking order.
`ConfigureRequest`	Generated before a change of 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).
`MapRequest`	Generated before a window is mapped.
`Motion`	The mouse is moving in the window.
`Leave`	The mouse is leaving the window.
`Map`	The window has been mapped (deiconified).
`Property`	A property on the window has been changed or deleted.
`Reparent`	A window has been reparented.
`ResizeRequest`	Generated before a window changes size.
`Unmap`	The window has been unmapped (iconified).
`Visibility`	The window has changed visibility.

Most of the events in the table have to do with dark corners of the X protocol. The most commonly used events are key presses, button presses, and mouse motion. The `Map` and `UnMap` events let an application respond to the open and close of the window. The `Configure` event is useful if the display needs to be changed when the window resizes. The remaining events are seldom used, and the request events are only designed for window manager applications.

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 it can be left off altogether if a detail is given to indicate what key. The following are all equivalent:

```
<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 <Any-Key> {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 inspite of the curly braces used for grouping. If the user types a capital `Q`, the output is:

```
%K=Shift_R %A="
%K=Q %A="Q"
```

The first line is not a misprint. `%A` is replaced with `NULL`, a zero-valued byte, when modifier keys are pressed. The `NULL` character terminates the enclosing string, just like `'\0'` terminates a string in C programs. This feature can be used in `<Any-Key>` bindings to avoid doing anything if a modifier key is pressed. The following might be used with a text widget.

```
bind $w <Any-Key> {
    if {%A != {}} {%W insert end %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 : a detail of `<1>` implies a `ButtonPress` event, while a detail of `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 <Any-Motion> {puts stdout "%W %x %y"}
```

MODIFIERS

A modifier indicates that another key or button is being held down at the time of the event. A table that summarizes the modifiers is given below.

Table 11–2 Event modifiers.

`Control`	The Control key.
`Shift`	The shift key.
`Lock`	The caps-lock key.
`Mod1, M1, Meta, M`	The meta key. The location and existence of the modifiers `Mod1` through `Mod5` is keyboard dependent.
`Mod2,M2,Alt`	The alt key.
`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).

Table 11–2 Event modifiers.

`Button3, B3`	The third mouse button (right).
`Button4, B4`	The fourth mouse button.
`Button5, B5`	The fifth mouse button.
`Any`	Match any modifier or combination of modifiers.
`Double`	Matches double press event.
`Triple`	Matches triple press event.

There are 8 modifier keys defined by the X protocol. The `Control`, `Shift`, and `Lock` modifiers are found on nearly all keyboards. `Mod1` is commonly known as the `Meta` key, and `Mod2` is commonly known as the `Alt` key. The remaining modifiers, `Mod3` through `Mod5`, are sometimes mapped to other special keys. The UNIX `xmodmap` program will return the current mappings from keys to these modifiers. It can also be used to change the mapping.

Example 11–1 Output from the UNIX `xmodmap` program. The first column lists the modifier. The rest of each line identifies the keysym(s) and low-level keycodes that are mapped to each modifier.

```
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 `Any` modifier is a wild card that matches if zero or more modifiers are in effect. If there is a binding for a more specific event, then the more specific binding is used. For example, `<Control-a>` is more specific than `<Any-a>`. `<Any-Key>` is a useful event that matches if there is no binding for a specific key press.

```
bind $w <Any-Key> {
     if {%A != ""} { Status "Unknown command %A" }
}
```

When a modifier key is pressed, a `KeyPress` event is generated but the printing character associated with it is `NULL`. This is tested in the `<Any-Key>` binding above so the warning is not generated when a modifier key is pressed.

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. Verify 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.*

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.

Example 11–2 Emacs-like binding convention for Meta and Escape. If there is a Meta key on the keyboard it is used like the Control or Shift modifier. In addition, the Escape key can be used in a sequence with the other key.

```
proc BindSequence { w seq cmd } {
    bind $w $seq $cmd
    # 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] {
```

* 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 is 500 milliseconds.

```
        bind $w $prefix { }
    }
}
```

Other Events

The <Map> and <Unmap> events are generated when a window is opened and closed, respectively.

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.)

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.

Table 11–3 A summary of the event keywords. 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.

%%	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) Above, Below, BottomIf, Opposite, TopIf. (ConfigureRequest)
%f	The focus field (0 or 1). (Enter, Leave)

Table 11–3 A summary of the event keywords. 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.

`%h`	The height field. (`Configure`, `ConfigureRequest`, `Expose`, `ResizeRequest`)
`%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`, `CirculateRequest`)
`%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`, `ConfigureRequest`, `Expose`, `ResizeRequest`)
`%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`)

A Binding User Interface

As a way to demonstrate bindings, a simple user interface to the binding mechanism is presented. The interface displays the current bindings in effect for a specified widget or class. A pair of `listbox` widgets are used to display the event sequences and the commands bound to each sequence. A `listbox` provides a line-oriented, scrollable text display . The interface lets the user edit existing bindings and define new bindings.

Example 11–3 A user interface to widget bindings.

```
proc Bind_Interface { frame } {
    # 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 -relief sunken
    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 BindRedisplay $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_Define $frame]
    pack $t.quit $t.save $t.edit $t.new -side right

    # A pair of listboxes to display the event sequences
    # and the commands bound to them.

    listbox $frame.key -geometry 18x5 \
        -yscrollcommand "$frame.s set" -relief raised
    listbox $frame.cmd -geometry 25x5 \
        -yscrollcommand "$frame.s set" -relief raised \
        -exportselection false
    scrollbar $frame.s -orient vertical \
        -command [list BindYview $frame]
    pack $frame.s -side left -fill y
    pack $frame.key $frame.cmd -side left \
        -fill both -expand true

    # Bindings so the listboxes scroll together.

    foreach l [list $frame.key $frame.cmd] {
        bind $l <Button-2> \
```

```
            [list BindMark %x %y $frame]
        bind $l <B2-Motion> \
            [list BindDragto %x %y $frame]
        bind $l <Button-1> \
            [list BindSelect %x %y $frame]
        bind $l <B1-Motion> \
            [list BindSelect %x %y $frame]
        global tk_version
        if {$tk_version <= 3.6} {
            bind $l <Shift-Button-1> " "
            bind $l <Shift-B1-Motion> " "
        } else { # Tk 4.*
            bind $l <Shift-Button-1> { break }
            bind $l <Shift-B1-Motion> { break }
        }
    }
}
```

The `Bind_Interface` command takes a frame as a parameter. It uses the frame to pack together the widgets in its display. If `frame` is specified as {}, then the widgets will be created as children of the main window. The `frame` parameter is used in several of the command bindings as a way to identify the binding interface. The possibility of a null frame value requires careful use of `list` in the definitions of the commands bound to buttons and events.

Compare the following bindings that recompute the display when the user presses `<Return>` in the entry widget.

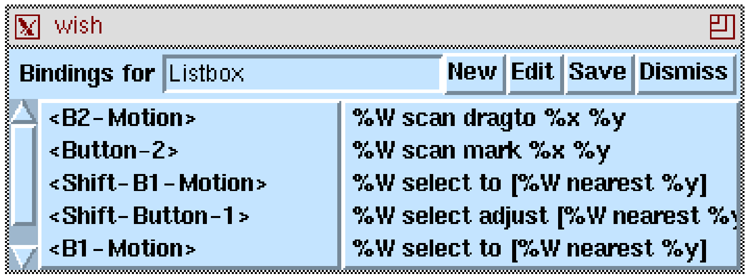
```
bind $t.e <Return> [list BindRedisplay $frame]
bind $t.e <Return> "BindRedisplay $frame"
```

In the second case, if `frame` has the value `{}`, then the use of double-quotes will `concat` things together and `BindRedisplay` will appear to have no arguments at all. The use of `list` to construct the command ensures that the argument structure is correct.

Example 11–4 Bind_Display presents the bindings for a given widget or class.

```
Bind_Display { frame w } {
    $frame.top.e delete 0 end
    $frame.top.e insert end $w
    $frame.key delete 0 end
    $frame.cmd delete 0 end
    foreach seq [bind $w] {
        $frame.key insert end $seq
        $frame.cmd insert end [bind $w $seq]
    }
}
```

Compare the text operations in the `entry` and `listbox` widgets. Both widgets use a single index, `0, 1, 2, ... end`, to represent positions in their text display. For an `entry` widget, however, it is a character position, and for the `listbox` it is a line position. Both count from zero and support the `end` keyword



`Bind_Display Listbox`

Two listboxes are created and configured to select items together . A `foreach` loop is used with the loop parameter taking on the names of the two listbox widgets. This avoids duplicating lots of commands, which can be tedious to maintain.

Example 11–5 Related listboxes are configured to select items together.

```
foreach l [list $frame.key $frame.cmd] {
    listbox $l -exportselection false \
        -yscrollcommand "$frame.s set" -relief raised
    bind $l <Button-1> [list BindSelect %x %y $frame]
    bind $l <B1-Motion> [list BindSelect %x %y $frame]
}

proc BindSelect { x y f } {
    foreach l {key cmd} {
        $f.$l select from [$f.$l nearest $y]
        $f.$l select to [$f.$l nearest $y]
    }
}
```

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 11–6 Controlling a pair of listboxes with one scrollbar.

```
scrollbar $frame.s -orient vertical \
    -command [list BindYview $frame]

proc BindYview { frame line } {
    foreach list {key cmd} {
        $frame.$list yview $line
    }
```

```
}
```

The BindYview command is the scrollbar's command that it uses to change the display of its associated widget, or widgets. Before the scroll command is evaluated an additional line parameter is added that specifies what line to position at the top of the display. This is done internally by the scrollbar implementation. BindYview uses the yview listbox operation to position both listboxes.

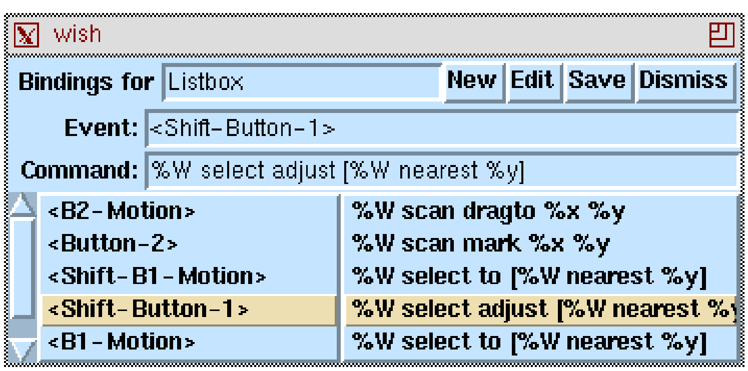
The Listbox class bindings for <Button-2> and <B2-Motion> cause the listbox to scroll as the user drags the widget with the middle mouse button. These bindings are adjusted in the example so that both listboxes move together.

Example 11–7 Drag-scrolling a pair of listboxes together.

```
bind $l <Button-2> [list BindMark %x %y $frame]
bind $l <B2-Motion> [list BindDragto %x %y $frame]

proc BindMark { x y frame } {
    foreach list {key cmd} {
        $frame.$list scan mark $x $y
    }
}
proc BindDragto { x y frame } {
    foreach list {key cmd}{
        $frame.$list scan dragto $x $y
    }
}
```

This is done with the scan mark listbox command, which defines an origin, and the scan dragto listbox command, which scrolls the widget based on the distance from that origin. All Tk widgets that scroll support yview, scan mark, and scan dragto.

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 11–8 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 -relief sunken
         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 -relief sunken -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 11–9 Defining and saving bindings.

```
proc BindDefine { f } {
   bind [$f.top.e get] [$f.edit.top.e get] \
      [$f.edit.bot.e get]
```

```
    pack forget $f.edit
}
proc Bind_Save { dotfile args } {
    set out [open $dotfile.new w]
    foreach w $args {
        foreach seq [bind $w] {
            puts $out [list bind $w $seq [bind $w $seq]]
        }
    }
    exec mv $dotfile.new $dotfile
}
proc Bind_Read { dotfile } {
    if [catch {
        source $dotfile
    } err] {
        Status "Bind_Read $dotfile failed: $err"
    }
}
```

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.

C H A P T E R 12

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 `TkSimpleApp`.

```
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 Tcl `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, userDefault, interactive`. These names can be abbreviated. The default priority is `interactive`.

Example 12–1 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 12–2 A file containing X resource specifications.

```
!
```

* 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).

```
! Grey color set
! Slightly modified from Ron Frederick's NV grey family
!
*activeBackground: white
*activeForeground: black
*selector: black
*background: #efefef
*foreground: black
*selectBackground: #bfdfff
*Scale.sliderForeground: #dfdfdf
*Scrollbar.foreground: #dfdfdf
*Scrollbar.activeForeground: white
*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.activeForeground white
```

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 12–3 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 multiword 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 12–4 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 12–5 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. Y ou 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 12–6 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]
					$menu add cascade -label $l -command $c \
						-menu $submenu
					MenuButtonInner $submenu
				}
			}
			default {
				$menu add command -label $l -command $c
			}
		}
	}
}
```

CHAPTER 13

Managing User Preferences

This chapter describes a user preferences package that is layered on top of X resources.

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 `Preferences_Init` procedure is called like this:

```
Preferences_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 13–1 Preferences Initialization.

```
proc Preferences_Init { userDefaults appDefaults } {
	global pref

	set pref(uid) 0;# for a unique identifier for widgets
	set pref(userDefaults) $userDefaults
	set pref(appDefaults) $appDefaults

	PreferencesReadFile $appDefaults startup
	PreferencesReadFile $userDefaults user
}
proc PreferencesReadFile { 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 `PreferencesReadFile` 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`. Note that Tk currently cannot tell the difference between color and greyscale because it just examines the number of bits in the display depth and treats anything with more than one bit-per-pixel as color.

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. You could extend `PreferencesReadFile` to allow for per-host files as well.

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 `Preferences_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 `Preferences_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. `Preferences_Add` is shown along with these below:

Example 13–2 Preferences_Add defines a set of 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 Preferences_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 Preferences_Add. If a user specifies a boolean resource manually, they might use “true” instead of 1 and “false” instead of 0. Preferences_Add could 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 13–3 PrefValue returns a global variable’s current value, or the value from the resource database. PrefValueSet sets a variable in the global scope.

```
proc PrefValue { varName xres } {
    upvar #0 $varName var
    if [info exists var] {
        return $var
    }
    set var [option get . $xres {}]
}
proc PrefValueSet { varName value } {
    upvar #0 $varName var
    set var $value
}
```

An important side effect of the Preferenes_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 13–4 Using Preferences_Add.

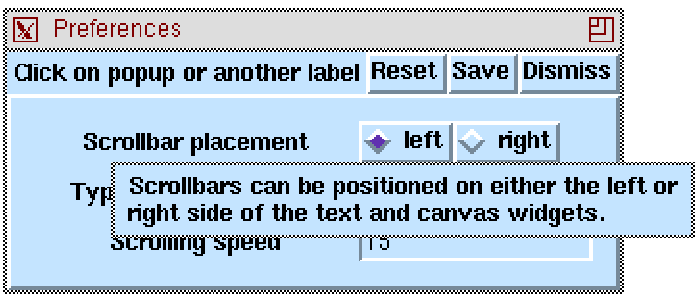
```
Preferences_Add {
    {win(scrollside) scrollbarSide {CHOICE left right}
        "Scrollbar placement"
"Scrollbars can be positioned on either the left or
right side of the text and canvas widgets."}
    {win(typeinkills) typeinKills OFF
        "Type-in kills selection"
"This setting determines 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 Preferences_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 Preferences_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 Preferences_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 Preferences_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 13–5 Preferences_Dialog constructs an interface to set the preference items.

```
proc Preferences_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 {PreferencesDismiss}
        button $buttons.save -text Save \
            -command {PreferencesSave}
        button $buttons.reset -text Reset \
            -command {PreferencesReset ; PreferencesDismiss}
        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) {
            PreferencesDialogItem $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 13–6 PreferencesDialogItem handles the different types of preferences.

```
proc PreferencesDialogItem { 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]
            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 13–7 PrefItemHelp displays 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 13–8 PreferencesSave writes the resource specifications to the end of the per-user resource file, allowing users to add other resources to the beginning.

```
proc PreferencesSave {} {
    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
    }
    PreferencesDismiss
}
```

There is one fine point in PreferencesSave, 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 13–9 PreferencesReset restores the settings from the preferences file.

```
proc PreferencesReset {} {
    global pref
    # Re-read user defaults
    option clear
    PreferencesReadFile $pref(appDefaults) startup
    PreferencesReadFile $pref(userDefaults) user
    # Clear variables
    set items $pref(items)
    set pref(items) {}
    foreach item $items {
        uplevel #0 [list unset [PrefVar $item]]
    }
    # Restore values
    Preferences_Add $items
}
proc PreferencesDismiss {} {
    destroy .pref
    catch {destroy .prefitemhelp}
}
```

The interface is rounded out with the PreferencesReset and Preferences-

Dismiss 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 Preferences_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 Preferences 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 Preference_Add is the only command using the variable.

Example 13–10 Tracing a Tcl variable in a preference item.

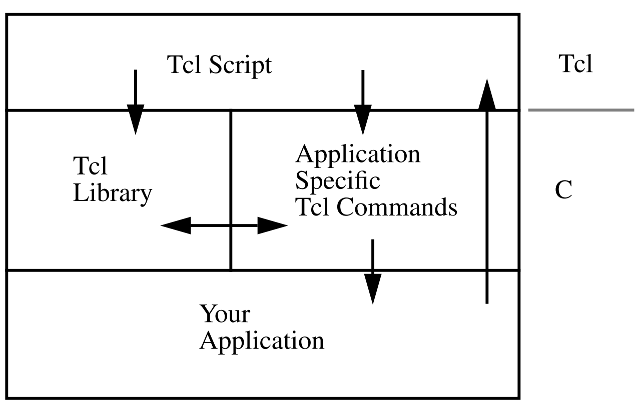
```
Preferences_Add {
    {win(scrollside) scrollbarSide {CHOICE left right}
        "Scrollbar placement"
"Scrollbars can be positioned on either the left or
right side of the text and canvas widgets."}
}
global win
set win(lastscrollside) $win(scrollside)
trace variable win(scrollside) w ScrollFixup

# Assume win(scrollbar) identifies the scrollbar widget

proc ScrollFixup { name1 name2 op } {
    global win
    if {$win(scrollside) != $win(lastscrollside)} {
        set parent [lindex [pack newinfo $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 14

C Programming and Tcl

This chapter describes the relationship between C extensions and the 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 will call 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.

The application creates and 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. At this point the application typically enters some processing 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.

USING THE STANDARD MAIN PROGRAMS

The basic application structure has been extracted into a standard main program that is part of both the Tcl and Tk C libraries. The main program takes care of the details described above and then calls Tcl_AppInit to complete the initialization of your application

Example 14–1 Tcl_AppInit is called by the standard main program in the Tcl library.

```
/* myAppInit.c */
#include <tcl.h>
/* Required on some systems to use main from library. */
extern int main();
int *tclDummyMainPtr = (int *) main;

int RandomCmd(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]);

int
Tcl_AppInit(Tcl_Interp *interp) {
    /*
     * Initialize packages.
     */
    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.
     */
    tcl_RcFileName = "~/.myapp.tcl";
    return TCL_OK;
```

```
}
```

The main program in the library calls Tcl_AppInit with one argument, a handle on a newly created interpreter*. There are three parts to the procedure. The first part initializes the various packages used by the application. While the core Tcl commands are defined during Tcl_CreateInterp, which is called by main, there is some additional setup required for the Tcl library facility that is done by the Tcl_Init call. Initialization consists of sourcing the init.tcl file (or complaining if it does not exists). The default init.tcl defines an unknown command that implements the library facility and auto-exec facilities.

The second part of Tcl_AppInit does application-specific initialization. The example registers a command procedure, RandomCmd that implements a new Tcl command, random. 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 next example shows a Tcl_AppInit that initializes several packages, but defines no other application-specific commands. Most of the packages available from the Tcl archive have been structured so you can initialize them like this. To create the full application, the Tcl_AppInit routine is linked with the libraries for Tcl and the various extensions being used. The main program from the Tcl (or Tk) library will be used, and it will call your Tcl_AppInit.

Example 14–2 Another Tcl_AppInit example.

```
/* myAppInit.c */
#include <stdio.h>
#include <tcl.h>
/* Required on some systems to use main from library. */
extern int main();
int *tclDummyMainPtr = (int *) main;

int
Tcl_AppInit(Tcl_Interp *interp) {
    /*
     * Initialize packages:
     * Tcl core (unknown handler)
     * Extended Tcl package
     * Tk widget library
     * Dp socket extension
     * Blt widgets and table geometry manager
     */
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (TclX_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
```

* Tcl_AppInit is not defined by the library so you get errors from the linker if you do not supply an implementation.

```
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Dp_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Blt_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

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.

Example 14–3 The `RandomCmd` C command procedure.

```
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 200 bytes. The default cleanup action is to do nothing. These defaults support two simple ways to define the result of a command. The first way is to set `interp->result` to the address of a constant string.

```
interp->result = "Usage: random ?random?";
```

The second way is to use sprintf to format the result in place.

```
sprintf(interp->result, "%d", rand);
```

In more general cases the `Tcl_SetResult`, `Tcl_AppendResult`, and `Tcl_AppendElement` 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. 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.

A Custom Main Program

In more complex applications you 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.

Example 14–4 A custom main program.

```
main(int argc, char *argv[])
{
    Tcl_Interp *interp;
    int error; char *trace;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(1);
    }
    interp = Tcl_CreateInterp();

    mainWindow = Tk_CreateMainWindow(interp,
        getenv("DISPLAY"), "myapp", "Tk");
    if (mainWindow == NULL) {
        fprintf(stderr, "%s\n", interp->result);
        exit(1);
    }
    Tk_GeometryRequest(mainWindow, 200, 200);

    /*
     * 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);
        }
    }
    while (MyappExists()) {
        Tk_DoOneEvent();
        MyappStuff();
    }
```

```
}
```

Tcl_CreateInterp creates an interpreter context. 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 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 #!.

The end of the main program has an event loop. This is a potentially tricky area. Tk can provide its own event loop that allows you to register handlers for I/O streams and other events (see Tcl_CreateFileHandler, Tcl_CreateIdleHandler, and Tcl_CreateTimerHandler). If you provide your own loop, you need to call Tk_DoOneEvent regularly to keep the user interface active.

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. After any Tcl command is evaluated from C, the result is available in the interp->result field. It contains either the result of the command if TCL_OK is returned, or an error message otherwise. Tcl_EvalFile is a variation on the basic Tcl_Eval call, and they all share this result protocol.

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. The variable might not be defined if the script simply didn't exist in the first place. In practice, you would probably prompt the user before dumping the Tcl trace.

PUTTING A TCL PROGRAM TOGETHER

Assuming you've put the simple Tcl_AppInit and RandomCmd from the examples into a file name cprog1.c, you are ready to try it 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 14–5 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 = cprog1.o

cprog1 : $(OBJS)
       $(CC) -o cprog1 $(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.

Tcl Shell Library Environment

It may also 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 call `Tcl_Init`, which does an `eval` of the following command:

```
source [info library]/init.tcl
```

The primary thing defined by `init.tcl` is the implementation of the `unknown` procedure. That defines the library implementation which allows a script to automatically load Tcl procedures from libraries.

For Tk, `wish` calls `Tcl_Init` and `Tk_Init`, which executes the following command.

```
source $tk_library/tk.tcl
```

It is crucial that the Tk library be set up properly because much of the default behavior for its widgets is defined there, not in the C implementation. So, one of the first things done by the `tk.tcl` script is to update the `auto_path`:

```
lappend auto_path $tk_library
```

To summarize, the bootstrap works as follows:

* You can also override these settings with environment variables), `TCL_LIBRARY` and `TK_LIBRARY`, but you shouldn't have to resort to that.

- 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.

N

O

P

R

S

T

X

Y