PART I

Tcl Basics

Part I introduces the basics of Tcl. Everyone should read Chapter 1, which describes the fundamental properties of the language. Tcl is really quite simple, so beginners can pick it up quickly. The experienced programmer should review Chapter 1 to eliminate any misconceptions that come from using other languages.

Chapter 2 is a short introduction to running Tcl and Tk on UNIX, Windows, and Macintosh systems. You may want to look at this chapter first so you can try out the examples as you read Chapter 1.

Chapter 3 presents a sample application, a CGI script, that implements a guestbook for a Web site. The example uses several facilities that are described in detail in later chapters. The goal is to provide a working example that illustrates the power of Tcl.

The rest of Part I covers basic programming with Tcl. Simple string processing is covered in Chapter 4. Tcl lists, which share the syntax rules of Tcl commands, are explained in Chapter 5. Control structure like loops and if statements are described in Chapter 6. Chapter 7 describes Tcl procedures, which are new commands that you write in Tcl. Chapter 8 discusses Tcl arrays. Arrays are the most flexible and useful data structure in Tcl. Chapter 9 describes file I/O and running other programs. These facilities let you build Tcl scripts that glue together other programs and process data in files.

After reading Part I you will know enough Tcl to read and understand other Tcl programs, and to write simple programs yourself.

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 Tcl syntax is meant to be simple. Tcl is designed to be a glue that assembles software building blocks into applications. A simpler glue makes the job easier. In addition, Tcl is interpreted when the application runs. The interpreter makes it easy to build and refine your application in an interactive manner. A great way to learn Tcl is to try out commands interactively. If you are not sure how to run Tcl on your system, see Chapter 2 for instructions for starting Tcl on UNIX, Windows, and Macintosh systems.

This chapter takes you through the basics of the Tcl language syntax. Even if you are an expert programmer, it is worth taking the time to read these few pages to make sure you understand the fundamentals of Tcl. 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 from some other programming languages with which you may already be familiar, so it is worth making sure you understand the basic concepts.

Tcl Commands

Tcl stands for Tool Command Language. A command does something for you, like output a string, compute a math expression, or display a widget on the screen. Tcl casts everything into the mold of a command, even programming constructs like variable assignment and procedure definition. Tcl adds a tiny amount of syntax needed to properly invoke commands, and then it leaves all the hard work up to the command implementation.

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 (i.e., spaces or tabs) is used to separate the command name and its arguments, and a newline (i.e., the end of line character) or semicolon is used to terminate a command. Tcl does not interpret the arguments to the commands except to perform *grouping*, which allows multiple words in one argument, and *substitution*, which is used with programming variables and nested command calls. The behavior of the Tcl command processor can be summarized in three basic steps:

- Argument grouping.
- Value substitution of nested commands, variables, and backslash escapes.
- Command invocation. It is up to the command to interpret its arguments.

This model is described in detail in this Chapter.

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. Chapter 9 describes how to open other files for I/O.
- Curly braces are used to group words together into a single argument. The `puts` command receives `Hello, World!` as its second argument.

The braces are not part of the value.

The braces are syntax for the interpreter, and they get stripped off before the value is passed to the command. Braces group all characters, including newlines and nested braces, until a matching brace is found. Tcl also uses double quotes for grouping. Grouping arguments will be described in more detail later.

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. In fact, you can use any character in a variable name.

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 in Example 1–2:

Example 1–2 Tcl variables.

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

The second `set` command 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 of substitution is more efficient, which is important when the value is large.

Command Substitution

The second form of substitution is *command substitution*. A nested command is delimited by square brackets, `[ ]`. 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 commands.

Example 1–3 Command substitution.

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

In Example 1–3, the nested command is:

```
string length foobar
```

This command returns the length of the string `foobar`. The `string` command is described in detail starting on page 45. The nested command runs first.

Then, 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. This results in a sensible ordering in which nested commands are evaluated first so that their result can be used in arguments to the outer command.

Math Expressions

The Tcl interpreter itself does not evaluate math expressions. Tcl just does grouping, substitutions and command invocations. The `expr` command is used to parse and evaluate math expressions.

Example 1–4 Simple arithmetic.

```
expr 7.2 / 4
=> 1.8
```

The math syntax supported by `expr` is the same as the C expression syntax. The `expr` command 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. Octal values are indicated by a leading zero (e.g., `033` is `27` decimal). Hexadecimal values are indicated by a leading `0x`. Scientific notation for floating point numbers is supported. A summary of the operator precedence is given on page 20.

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

Example 1–5 Nested commands.

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

The expression evaluator supports a number of built-in math functions. (For a complete listing, see page 21.) Example 1–6 computes the value of *pi*:

Example 1–6 Built-in math functions.

```
set pi [expr 2*asin(1.0)]
=> 3.1415926535897931
```

The implementation of `expr` is careful to preserve accurate numeric values and avoid conversions between numbers and strings. However, you can make `expr` operate more efficiently by grouping the entire expression in curly braces. The explanation has to do with the byte code compiler that Tcl uses internally, and its effects are explained in more detail on page 15. For now, you should be aware that these expressions are all valid and run a bit faster than the examples shown above:

Example 1–7 Grouping expressions with braces.

```
expr {7.2 / 4}
set len [expr {[string length foobar] + $x}]
set pi [expr {2*asin(1.0)}]
```

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. 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. In particular, the `list` command described on page 61 will do quoting for you automatically. In Example 1–8 backslash is used to get a literal `$`:

Example 1–8 Quoting special characters with backslash.

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

Only a single round of interpretation is done.

The second `set` command in the example illustrates an important property of Tcl. The value of `dollar` does not affect the substitution performed in the assignment to `x`. In other words, the Tcl parser does not care about the value of a variable when it does the substitution. In the example, the value of `x` and `dollar` is the string `$foo`. In general, you do not have to worry about the value of variables until you use `eval`, which is described in Chapter 10.

You can also use backslash sequences to specify characters with their Unicode, hexadecimal, or octal value:

```
set escape \u001b
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 20 summarizes backslash substitutions.

A common use of backslashes is to continue long commands on multiple lines. This is necessary because a newline terminates a command. The backslash in the next example is required; otherwise the `expr` command gets terminated by the newline after the plus sign.

Example 1–9 Continuing long lines with backslashes.

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

There are two fine points to escaping newlines. First, if you are grouping an argument as described in the next section, then you do not need to escape newlines; the newlines are automatically part of the group and do not terminate the command. Second, a backslash as the last character in a line is converted into a space, and all the white space at the beginning of the next line is replaced by this substitution. In other words, the backslash-newline sequence also consumes all the leading white space on the next line.

Grouping with Braces and Double Quotes

Double quotes and curly braces are used to group words together into one argument. The difference between double quotes and curly braces is that quotes allow substitutions to occur in the group, while curly braces prevent substitutions. This rule applies to command, variable, and backslash substitutions.

Example 1–10 Grouping with double quotes vs. braces.

```
set s Hello
=> 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 second command of Example 1–10, the Tcl interpreter does variable and command substitution on the second argument to `puts`. In the third command, substitutions are prevented, so the string is printed as is.

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

Another common use of quotes is with the `format` command. This 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 easiest way to specify these characters is with backslash sequences (e.g., `\n` for newline and `\t` for tab). The backslashes must be substituted before the `format` command is called, so you need to use quotes to group the format specifier.

```
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. For more information about the `format` command, see page 52.

Square Brackets Do Not Group

The square bracket syntax used for command substitution does not provide grouping. Instead, a nested command is considered part of the current group. In the command below, the double quotes group the last argument, and the nested command is just part of that group.

```
puts stdout "The length of $s is [string length $s]."
```

If an argument is made up of only a nested command, you do not need to group it with double-quotes because the Tcl parser treats the whole nested command as part of the group.

```
puts stdout [string length $s]
```

The following is a redundant use of double quotes:

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

Grouping before Substitution

The Tcl parser makes a single pass through a command as it makes grouping decisions and performs string substitutions. Grouping decisions are made before substitutions are performed, which is an important property of Tcl. This means that the values being substituted do not affect grouping because the grouping decisions have already been made.

The following example demonstrates how nested command substitution affects grouping. A nested command is treated as an unbroken sequence of characters, regardless of its internal structure. It is included with the surrounding group of characters when collecting arguments for the main command.

Example 1–11 Embedded command and variable substitution.

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

In Example 1–11, 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. By the time Tcl encounters the left bracket, it has already done some variable substitutions to obtain:

```
7+9=
```

When the left bracket is encountered, the interpreter 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
```

Grouping before substitution.

The point of this example 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. Grouping and variable substitution interact the same as grouping and command substitution. Spaces or special characters in variable values do not affect grouping decisions because these decisions are made before the variable values are substituted.

If you want the output to look nicer in the example, with spaces around the `+` and `=`, then you must use double quotes to explicitly group the argument to `puts`:

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

The double quotes are used for grouping in this case to allow the variable and command substitution on the argument to `puts`.

Grouping Math Expressions with Braces

It turns out that `expr` does its own substitutions inside curly braces. This is explained in more detail on page 15. This means you can write commands like the one below and the substitutions on the variables in the expression still occur:

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

More Substitution Examples

If you have several substitutions with no white space between them, you can avoid grouping with quotes. The following command sets `concat` to the value of variables `a`, `b`, and `c` all concatenated together:

```
set concat $a$b$c
```

Again, if you want to add spaces, you'll need to use quotes:

```
set concat "$a $b $c"
```

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

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

When you use Tk, you often use widget names as command names:

```
$text insert end "Hello, World!"
```

Procedures

Tcl uses the proc command to define procedures. Once defined, a Tcl procedure is used just like any of the other built-in Tcl commands. The basic syntax to define a procedure is:

```
proc name arglist body
```

The first argument is the name of the procedure being defined. The second argument is a list of parameters to the procedure. The third argument is a *command body* that is one or more Tcl commands.

The procedure name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each other. As a convention, this book begins procedure names with uppercase letters and it begins variable names with lowercase letters. Good programming style is important as your Tcl scripts get larger. Tcl coding style is discussed in Chapter 12.

Example 1–12 Defining a procedure.

```
proc Diag {a b} {
   set c [expr sqrt($a * $a + $b * $b)]
   return $c
}
puts "The diagonal of a 3, 4 right triangle is [Diag 3 4]"
=> The diagonal of a 3, 4 right triangle is 5.0
```

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 defined only during execution of Diag. Variable scope is discussed further in Chapter 7. It is not really necessary to use the variable c in this example. The procedure can also be written as:

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

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

```
proc Diag {a b} {
     expr sqrt($a * $a + $b * $b)
}
```

Note the stylized use of curly braces in the example. The curly brace at the end of the first line starts the third argument to proc, which is the command body. In this case, the Tcl interpreter sees the opening left brace, causing it to ignore newline characters and scan the 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 7.

A Factorial Example

To reinforce what we have learned so far, below is a longer example that uses a `while` loop to compute the factorial function:

Example 1–13 A `while` loop to compute factorial.

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

The semicolon is used on the first line to remind you that it is a command terminator just like the newline character. The `while` loop is used to multiply all the numbers from one up to the value of `x`. The first argument to `while` is a boolean expression, and its second argument is a command body to execute. The `while` command and other control structures are described in Chapter 6.

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 to `while`, even if you have a much more complex expression.

The loop body and the procedure body are grouped with curly braces in the same way. The opening curly brace must be on the same line as `proc` and `while`. If you like to put opening curly braces on the line after a `while` or `if` statement, you must escape the newline with a backslash:

```
while {$i < $x} \
{
   set product ...
}
```

Always group expressions and command bodies with curly braces.

Curly braces around the boolean expression are crucial because they delay 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 reason 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 groups expressions with curly braces:

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

The incr command is used to increment the value of the loop variable i. This is a handy command that saves us from the longer command:

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

The incr command can take an additional argument, a positive or negative integer by which to change the value of the variable. Using this form, it is possible to eliminate the loop variable i and just modify the parameter x. The loop body can be written like this:

```
while {$x > 1} {
     set product [expr $product * $x]
     incr x -1
}
```

Example 1–14 shows factorial again, this time using a recursive definition. A recursive function is one that calls itself to complete its work. Each recursive call decrements x by one, and when x is one, then the recursion stops.

Example 1–14 A recursive definition of factorial.

```
proc Factorial {x} {
   if {$x <= 1} {
      return 1
   } else {
      return [expr $x * [Factorial [expr $x - 1]]]
   }
}
```

More about Variables

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 an easy way to use the set command. Example 1–15 shows a trick you can play by putting the name of one variable into another variable:

* Ironically, Tcl 8.0 introduced a byte-code compiler, and the first releases of Tcl 8.0 had a bug in the compiler that caused this loop to terminate! This bug is fixed in the 8.0.5 patch release.

Example 1–15 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`. Nested `set` commands provide another way to achieve a level of indirection. The last `set` command above can be written as follows:

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

Using a variable to store the name of another variable may seem overly complex. However, there are some times when it is very useful. There is even a special command, `upvar`, that makes this sort of trick easier. The `upvar` command is described in detail in Chapter 7.

Funny Variable Names

The Tcl interpreter makes some assumptions about variable names that make it easy to embed variable references into other strings. By default, it assumes that variable names contain only 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 white space, then you can use curly braces to explicitly delimit the variable name (e.g., `${x}`). You can also use this to reference variables with funny characters in their name, although you probably do not want variables named like that. If you find yourself using funny variable names, or computing the names of variables, then you may want to use the `upvar` command.

Example 1–16 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 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.

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 exist, you might have to test for the existence of the variable first.

Example 1–17 Using info to determine if a variable exists.

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

Example 7–6 on page 86 implements a new version of incr which handles this case.

More about Math Expressions

This section describes a few fine points about math in Tcl scripts. In Tcl 7.6 and earlier versions math is not that efficient because of conversions between strings and numbers. The expr command must convert its arguments from strings to numbers. It then does all its computations with double precision floating point values. The result is formatted into a string that has, by default, 12 significant digits. This number can be changed by setting the tcl_precision variable to the number of significant digits desired. Seventeen digits of precision are enough to ensure that no information is lost when converting back and forth between a string and an IEEE double precision number:

Example 1–18 Controlling precision with tcl_precision.

```
expr 1 / 3
=> 0
expr 1 / 3.0
=> 0.333333333333
set tcl_precision 17
=> 17
expr 1 / 3.0
# The trailing 1 is the IEEE rounding digit
=> 0.33333333333333331
```

In Tcl 8.0 and later versions, the overhead of conversions is eliminated in most cases by the built-in compiler. Even so, Tcl was not designed to support math-intensive applications. You may want to implement math-intensive code in a compiled language and register the function as a Tcl command as described in Chapter 44.

There is support for string comparisons by `expr`, so you can test string values in `if` statements. You must use quotes so that `expr` knows to do string comparisons:

```
if {$answer == "yes"} { ... }
```

However, the `string compare` and `string equal` commands described in Chapter 4 are more reliable because `expr` may do conversions on strings that look like numbers. The issues with string operations and `expr` are discussed on page 48.

Expressions can include variable and command substitutions and still be grouped with curly braces. This is because an argument to `expr` is subject to two rounds of substitution: one by the Tcl interpreter, and a second by `expr` itself. Ordinarily this is not a problem because math values do not contain the characters that are special to the Tcl interpreter. The second round of substitutions is needed to support commands like `while` and `if` that use the expression evaluator internally.

Grouping expressions can make them run more efficiently.

You should always group expressions in curly braces and let `expr` do command and variable substitutions. Otherwise, your values may suffer extra conversions from numbers to strings and back to numbers. Not only is this process slow, but the conversions can loose precision in certain circumstances. For example, suppose `x` is computed from a math function:

```
set x [expr {sqrt(2.0)}]
```

At this point the value of `x` is a double-precision floating point value, just as you would expect. If you do this:

```
set two [expr $x * $x]
```

then you may or may not get 2.0 as the result! This is because Tcl will substitute `$x` and `expr` will concatenate all its arguments into one string, and then parse the expression again. In contrast, if you do this:

```
set two [expr {$x * $x}]
```

then `expr` will do the substitutions, and it will be careful to preserve the floating point value of `x`. The expression will be more accurate and run more efficiently because no string conversions will be done. The story behind Tcl values is described in more detail in Chapter 44 on C programming and Tcl.

Comments

Tcl uses the pound character, `#`, for comments. Unlike in many other languages, the # must occur at the beginning of a command. A # that occurs elsewhere is not treated specially. An easy trick to append a comment to the end of a command is to precede the # with a semicolon 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 for is that a backslash effectively continues a comment line onto the next line of the script. In addition, a semicolon 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
```

The behavior of a backslash in comments is pretty obscure, but it can be exploited as shown in Example 2–3 on page 27.

A surprising property of Tcl comments is that curly braces inside comments are still counted for the purposes of finding matching brackets. I think the motivation for this mis-feature was to keep the original Tcl parser simpler. However, it means that the following will not work as expected to comment out an alternate version of an if expression:

```
# if {boolean expression1} {
if {boolean expression2} {
    some commands
}
```

The previous sequence results in an extra left curly brace, and probably a complaint about a missing close brace at the end of your script! A technique I use to comment out large chunks of code is to put the code inside an `if` block that will never execute:

```
if {0} {
unused code here
}
```

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:

- Command arguments are separated by white space, unless arguments are grouped with curly braces or double quotes as described below.
- 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, semicolons, and nested braces. The enclosing (i.e., outermost) braces are not included in the group's value.
- Grouping with double quotes, `" "`, allows substitutions. The interpreter groups everything until another double quote is found, including newlines and semicolons. The enclosing quotes are not included in the group of char-

acters. A double-quote character can be included in the group by quoting it with a backslash, (e.g., \").

- Grouping decisions are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A dollar sign, $, causes variable substitution. Variable names can be any length, and case is significant. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the ${*varname*} syntax.
- 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.
- 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 or group of characters are replaced with a new character.
- Substitutions can occur anywhere unless prevented by curly brace grouping. Part of a group 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.
- A single round of substitutions is performed before command invocation. 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 do not have to worry about special characters in values causing extra substitutions.

Fine Points

- A common error is to forget a space between arguments when grouping with braces or quotes. This is because white space is used as the separator, while the braces or quotes only provide grouping. If you forget the space, you will get syntax errors about unexpected characters after the closing brace or quote. The following is an error because of the missing space between } and {:

```
if {$x > 1}{puts "x = $x"}
```

- A double quote is only used for grouping when it comes after white space. This means you can include a double quote in the middle of a group without quoting it with a backslash. This requires that curly braces or white space delimit the group. I do not recommend using this obscure feature, but this is what it looks like:

```
set silly a"b
```

- When double quotes are used for grouping, the special effect of curly braces is turned off. Substitutions occur everywhere inside a group formed with double quotes. In the next command, the variables are still substituted:

```
set x xvalue
set y "foo {$x} bar"
=> foo {xvalue} bar
```

- When double quotes are used for grouping and a nested command is encountered, the nested command can use double quotes for grouping, too.

```
puts "results [format "%f %f" $x $y]"
```

- Spaces are *not* required around the square brackets used for command substitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the current group. The following sets `x` to the concatenation of two command results because there is no space between `]` and `[`.

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

- Newlines and semicolons are ignored when grouping with braces or double quotes. They get included in the group of characters just like all the others. The following sets `x` to a string that contains newlines:

```
set x "This is line one.
This is line two.
This is line three."
```

- During command substitution, newlines and semicolons *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. This was illustrated in Example 1–9 on page 8.
- A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following sets `x` to the single character `$`.

```
set x $
```

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 the newline and the leading white space on the next line with a space.
`\\`	Backslash. ('\')
`\ooo`	Octal specification of character code. 1, 2, or 3 digits.
`\xhh`	Hexadecimal specification of character code. 1 or 2 digits.
`\uhhhh`	Hexadecimal specification of a 16-bit Unicode character value. 4 hex digits.
`\c`	Replaced with literal `c` if `c` is not one of the cases listed above. In particular, `\$`, `\"`, `\{`, `\}`, `\]`, and `\[` are used to obtain these characters.

Arithmetic Operators

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

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

Built-in Math Functions

Table 1–3 Built-in math functions.

`acos(x)`	Arccosine of *x*.
`asin(x)`	Arcsine of *x*.
`atan(x)`	Arctangent 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.
`rand()`	Return a random floating point value between 0.0 and 1.0.
`srand(x)`	Set the seed for the random number generator to the integer *x*.

Core Tcl Commands

The pages listed in Table 1–4 give the primary references for the command.

Table 1–4 Built-in Tcl commands.

Command	Pg.	Description
`after`	218	Schedule a Tcl command for later execution.
`append`	51	Append arguments to a variable’s value. No spaces added.
`array`	91	Query array state and search through elements.
`binary`	54	Convert between strings and binary data.
`break`	77	Exit loop prematurely.
`catch`	77	Trap errors.
`cd`	115	Change working directory.
`clock`	173	Get the time and format date strings.
`close`	115	Close an open I/O stream.
`concat`	61	Concatenate arguments with spaces between. Splices lists.
`console`	28	Control the console used to enter commands interactively.
`continue`	77	Continue with next loop iteration.
`error`	79	Raise an error.
`eof`	109	Check for end of file.
`eval`	122	Concatenate arguments and evaluate them as a command.
`exec`	99	Fork and execute a UNIX program.
`exit`	116	Terminate the process.
`expr`	6	Evaluate a math expression.
`fblocked`	223	Poll an I/O channel to see if data is ready.
`fconfigure`	221	Set and query I/O channel properties.
`fcopy`	237	Copy from one I/O channel to another.
`file`	102	Query the file system.
`fileevent`	219	Register callback for event-driven I/O.
`flush`	109	Flush output from an I/O stream’s internal buffers.
`for`	76	Loop construct similar to C `for` statement.
`foreach`	73	Loop construct over a list, or lists, of values.
`format`	52	Format a string similar to C `sprintf`.
`gets`	112	Read a line of input from an I/O stream.
`glob`	115	Expand a pattern to matching file names.
`global`	84	Declare global variables.

Table 1–4 Built-in Tcl commands. (Continued)

`history`	185	Use command-line history.
`if`	70	Test a condition. Allows `else` and `elseif` clauses.
`incr`	12	Increment a variable by an integer amount.
`info`	176	Query the state of the Tcl interpreter.
`interp`	274	Create additional Tcl interpreters.
`join`	65	Concatenate list elements with a given separator string.
`lappend`	61	Add elements to the end of a list.
`lindex`	63	Fetch an element of a list.
`linsert`	64	Insert elements into a list.
`list`	61	Create a list out of the arguments.
`llength`	63	Return the number of elements in a list.
`load`	607	Load shared libraries that define Tcl commands.
`lrange`	63	Return a range of list elements.
`lreplace`	64	Replace elements of a list.
`lsearch`	64	Search for an element of a list that matches a pattern.
`lsort`	65	Sort a list.
`namespace`	203	Create and manipulate namespaces.
`open`	110	Open a file or process pipeline for I/O.
`package`	165	Provide or require code packages.
`pid`	116	Return the process ID.
`proc`	81	Define a Tcl procedure.
`puts`	112	Output a string to an I/O stream.
`pwd`	115	Return the current working directory.
`read`	113	Read blocks of characters from an I/O stream.
`regexp`	148	Match regular expressions.
`regsub`	152	Substitute based on regular expressions.
`rename`	82	Change the name of a Tcl command.
`return`	80	Return a value from a procedure.
`scan`	54	Parse a string according to a format specification.
`seek`	114	Set the seek offset of an I/O stream.
`set`	5	Assign a value to a variable.

Table 1–4 Built-in Tcl commands. (Continued)

`socket`	226	Open a TCP/IP network connection.
`source`	26	Evaluate the Tcl commands in a file.
`split`	65	Chop a string up into list elements.
`string`	45	Operate on strings.
`subst`	132	Substitute embedded commands and variable references.
`switch`	71	Test several conditions.
`tell`	114	Return the current seek offset of an I/O stream.
`time`	191	Measure the execution time of a command.
`trace`	183	Monitor variable assignments.
`unknown`	167	Handle unknown commands.
`unset`	13	Delete variables.
`uplevel`	130	Execute a command in a different scope.
`upvar`	85	Reference a variable in a different scope.
`variable`	197	Declare namespace variables.
`vwait`	220	Wait for a variable to be modified.
`while`	73	Loop until a boolean expression is false.

CHAPTER 2

Getting Started

This chapter explains how to run Tcl and Tk on different operating system platforms: UNIX, Windows, and Macintosh. Tcl commands discussed are: `source, console` and `info`.

This chapter explains how to run Tcl scripts on different computer systems. While you can write Tcl scripts that are portable among UNIX, Windows, and Macintosh, the details about getting started are different for each system. If you are looking for a current version of Tcl/Tk, check the Internet sites listed in the Preface on page *lii*.

The main Tcl/Tk program is *wish*. *Wish* stands for windowing shell, and with it you can create graphical applications that run on all these platforms. The name of the program is a little different on each of the UNIX, Windows, and Macintosh systems. On UNIX it is just *wish*. On Windows you will find *wish.exe*, and on the Macintosh the application name is *Wish*. A version number may also be part of the name, such as *wish4.2*, *wish80.exe*, or *Wish 8.2*. The differences among versions are introduced on page *xlviii*, and described in more detail in Part VII of the book. This book will use *wish* to refer to all of these possibilities.

Tk adds Tcl commands that are used to create graphical user interfaces, and Tk is described in Part III. You can run Tcl without Tk if you do not need a graphical interface, such as with the CGI script discussed in Chapter 3. In this case the program is *tclsh*, *tclsh.exe* or *Tclsh*.

When you run *wish,* it displays an empty window and prompts for a Tcl command with a % prompt. You can enter Tcl commands interactively and experiment with the examples in this book. On Windows and Macintosh, a console window is used to prompt for Tcl commands. On UNIX, your terminal window is used. As described later, you can also set up standalone Tcl/Tk scripts that are self-contained applications.

The source Command

You can enter Tcl commands interactively at the % prompt. It is a good idea to try out the examples in this book as you read along. The highlighted examples from the book are on the CD-ROM in the `exsource` folder. You can edit these scripts in your favorite editor. Save your examples to a file and then execute them with the Tcl `source` command:

```
source filename
```

The `source` command reads Tcl commands from a file and evaluates them just as if you had typed them interactively.

Chapter 3 develops a sample application. To get started, just open an editor on a file named `cgi1.tcl`. Each time you update this file you can save it, reload it into Tcl with the `source` command, and test it again. Development goes quickly because you do not wait for things to compile!

UNIX Tcl Scripts

On UNIX you can create a standalone Tcl or Tcl/Tk script much like an `sh` or `csh` script. The trick is in the first line of the file that contains your script. If the first line of a file begins with `#!pathname`, then UNIX uses `pathname` as the interpreter for the rest of the script. The "Hello, World!" program from Chapter 1 is repeated in Example 2–1 with the special starting line:

Example 2–1 A standalone Tcl script on UNIX.

```
#!/usr/local/bin/tclsh
puts stdout {Hello, World!}
```

Similarly, the Tk hello world program from Chapter 21 is shown in Example 2–2:

Example 2–2 A standalone Tk script on UNIX.

```
#!/usr/local/bin/wish
button .hello -text Hello -command {puts "Hello, World!"}
pack .hello -padx 10 -pady 10
```

The actual pathnames for *tclsh* and *wish* may be different on your system. If you type the pathname for the interpreter wrong, you receive a confusing "command not found" error. You can find out the complete pathname of the Tcl interpreter with the `info nameofexecutable` command. This is what appears on my system:

```
info nameofexecutable
=> /home/welch/install/solaris/bin/tclsh8.2
```

Watch out for long pathnames.

On most UNIX systems, this special first line is limited to 32 characters, including the `#!`. If the pathname is too long, you may end up with `/bin/sh` trying to interpret your script, giving you syntax errors. You might try using a symbolic link from a short name to the true, long name of the interpreter. However, watch out for systems like Solaris in which the script interpreter cannot be a symbolic link. Fortunately, Solaris doesn't impose a 32-character limit on the pathname, so you can just use a long pathname.

The next example shows a trick that works around the pathname length limitation in all cases. The trick comes from a posting to `comp.lang.tcl` by Kevin Kenny. It takes advantage of a difference between comments in Tcl and the Bourne shell. Tcl comments are described on page 16. In the example, the Bourne shell command that runs the Tcl interpreter is hidden in a comment as far as Tcl is concerned, but it is visible to `/bin/sh`:

Example 2–3 Using `/bin/sh` to run a Tcl script.

```
#!/bin/sh
# The backslash makes the next line a comment in Tcl \
exec /some/very/long/path/to/wish "$0" ${1+"$@"}
#   ... Tcl script goes here ...
```

You do not even have to know the complete pathname of *tclsh* or *wish* to use this trick. You can just do the following:

```
#!/bin/sh
# Run wish from the users PATH \
exec wish -f "$0" ${1+"$@"}
```

The drawback of an incomplete pathname is that many sites have different versions of *wish* and *tclsh* that correspond to different versions of Tcl and Tk. In addition, some users may not have these programs in their PATH.

If you have Tk version 3.6 or earlier, its version of *wish* requires a `-f` argument to make it read the contents of a file. The `-f` switch is ignored in Tk 4.0 and higher versions. The `-f`, if required, is also counted in the 32-character limit on `#!` lines.

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

Windows 95 Start Menu

You can add your Tcl/Tk programs to the Windows start menu. The command is the complete name of the *wish.exe* program and the name of the script. The trick is that the name of *wish.exe* has a space in it in the default configuration, so you must use quotes. Your start command will look something like this:

```
"c:\Program Files\TCL82\wish.exe" "c:\My Files\script.tcl"
```

This starts `c:\My Files\script.tcl` as a standalone Tcl/Tk program.

The Macintosh and *ResEdit*

If you want to create a self-contained Tcl/Tk application on Macintosh, you must copy the *Wish* program and add a Macintosh resource named `tclshrc` that has the start-up Tcl code. The Tcl code can be a single `source` command that reads your script file. Here are step-by-step instructions to create the resource using *ResEdit*:

- First, make a copy of *Wish* and open the copy in *ResEdit*.
- Pull down the `Resource` menu and select `Create New Resource` operation to make a new `TEXT` resource.
- *ResEdit* opens a window and you can type in text. Type in a `source` command that names your script:

```
source "Hard Disk:Tcl/Tk 8.1:Applications:MyScript.tcl"
```

- Set the name of the resource to be `tclshrc`. You do this through the `Get Resource Info` dialog under the `Resources` menu in *ResEdit*.

This sequence of commands is captured in an application called *Drag n Drop Tclets*, which comes with the Macintosh Tcl distribution. If you drag a Tcl script onto this icon, it will create a copy of *Wish* and create the `tclshrc` text resource that has a `source` command that will load that script.

If you have a Macintosh development environment, you can build a version of *Wish* that has additional resources built right in. You add the resources to the `applicationInit.r` file. If a resource contains Tcl code, you use it like this:

```
source -rcrc resource
```

If you don't want to edit resources, you can just use the *Wish* `Source` menu to select a script to run.

The `console` Command

The Windows and Macintosh platforms have a built-in console that is used to enter Tcl commands interactively. You can control this console with the `console` command. The console is visible by default. Hide the console like this:

```
console hide
```

Display the console like this:

```
console show
```

The console is implemented by a second Tcl interpreter. You can evaluate Tcl commands in that interpreter with:

```
console eval command
```

There is an alternate version of this console called *TkCon*. It is included on the CD-ROM, and you can find current versions on the Internet. *TkCon* was created by Jeff Hobbs and has lots of nice features. You can use *TkCon* on Unix systems, too.

Command-Line Arguments

If you run a script from the command line, for example from a UNIX shell, you can pass the script command-line arguments. You can also specify these arguments in the shortcut command in Windows. For example, under UNIX you can type this at a shell:

```
% myscript.tcl arg1 arg2 arg3
```

In Windows, you can have a shortcut that runs *wish* on your script and also passes additional arguments:

```
"c:\Program Files\TCL82\wish.exe" c:\your\script.tcl arg1
```

The Tcl shells pass the command-line arguments to the script as the value of the `argv` variable. The number of command-line arguments is given by the `argc` variable. The name of the program, or script, is not part of `argv` nor is it counted by `argc`. Instead, it is put into the `argv0` variable. Table 2–2 lists all the predefined variables in the Tcl shells. `argv` is a list, so you can use the `lindex` command, which is described on page 59, to extract items from it:

```
set arg1 [lindex $argv 0]
```

The following script prints its arguments (`foreach` is described on page 73):

Example 2–4 The `EchoArgs` script.

```
# Tcl script to echo command line arguments
puts "Program: $argv0"
puts "Number of arguments: $argc"
set i 0
foreach arg $argv {
   puts "Arg $i: $arg"
   incr i
}
```

Command-Line Options to *Wish*

Some command-line options are interpreted by *wish*, and they do not appear in the `argv` variable. The general form of the *wish* command line is:

```
wish ?options? ?script? ?arg1 arg2?
```

If no script is specified, then *wish* just enters an interactive command loop. Table 2–1 lists the options that *wish* supports:

Table 2–1 Wish command line options.

`-colormap new`	Use a new private colormap. See page 538.
`-display` *`display`*	Use the specified X *`display`*. UNIX only.
`-geometry` *`geometry`*	The size and position of the window. See page 570.
`-name` *`name`*	Specify the Tk application name. See page 560.

Table 2–1 Wish command line options. (Continued)

`-sync`	Run X synchronously. UNIX only.
`-use` *`id`*	Use the window specified by *`id`* for the main window. See page 578.
`-visual` *`visual`*	Specify the visual for the main window. See page 538.
`--`	Terminate options to *wish*.

Predefined Variables

Table 2–2 Variables defined by *tclsh* and *wish*.

`argc`	The number of command-line arguments.
`argv`	A list of the command-line arguments.
`argv0`	The name of the script being executed. If being used interactively, `argv0` is the name of the shell program.
`embed_args`	The list of arguments in the `<EMBED>` tag. Tcl applets only. See page 296.
`env`	An array of the environment variables. See page 117.
`tcl_interactive`	True (one) if the *tclsh* is prompting for commands.
`tcl_library`	The script library directory.
`tcl_patchLevel`	Modified version number, e.g., 8.0b1.
`tcl_platform`	Array containing operating system information. See page 182.
`tcl_prompt1`	If defined, this is a command that outputs the prompt.
`tcl_prompt2`	If defined, this is a command that outputs the prompt if the current command is not yet complete.
`tcl_version`	Version number.
`auto_path`	The search path for script library directories. See page 162.
`auto_index`	A map from command name to a Tcl command that defines it.
`auto_noload`	If set, the library facility is disabled.
`auto_noexec`	If set, the auto execute facility is disabled.
`geometry`	(*wish* only). The value of the `-geometry` argument.

CHAPTER 3

The Guestbook CGI Application

This chapter presents a simple Tcl program that computes a Web page. The chapter provides a brief background to HTML and the CGI interface to Web servers.

This chapter presents a complete, but simple guestbook program that computes an HTML document, or Web page, based on the contents of a simple database. The basic idea is that a user with a Web browser visits a page that is computed by the program. The details of how the page gets from your program to the user with the Web browser vary from system to system. The Tcl Web Server described in Chapter 18 comes with this guestbook example already set up. You can also use these scripts on your own Web server, but you will need help from your Webmaster to set things up.

The chapter provides a very brief introduction to HTML and CGI programming. HTML is a way to specify text formatting, including hypertext links to other pages on the World Wide Web. CGI is a standard for communication between a Web server that delivers documents and a program that computes documents for the server. There are many books on these subjects alone. *CGI Developers Resource, Web Programming with Tcl and Perl* by John Ivler (Prentice Hall, 1997) is a good reference for details that are left unexplained here.

A guestbook is a place for visitors to sign their name and perhaps provide other information. We will build a guestbook that takes advantage of the World Wide Web. Our guests can leave their address as a Universal Resource Location (URL). The guestbook will be presented as a page that has hypertext links to all these URLs so that other guests can visit them. The program works by keeping a simple database of the guests, and it generates the guestbook page from the database.

The Tcl scripts described in this chapter use commands and techniques that are described in more detail in later chapters. The goal of the examples is to demonstrate the power of Tcl without explaining every detail. If the examples in this chapter raise questions, you can follow the references to examples in other chapters that do go into more depth.

A Quick Introduction to HTML

Web pages are written in a text markup language called HTML (HyperText Markup Language). The idea of HTML is that you annotate, or mark up, regular text with special tags that indicate structure and formatting. For example, the title of a Web page is defined like this:

```
<TITLE>My Home Page</TITLE>
```

The tags provide general formatting guidelines, but the browsers that display HTML pages have freedom in how they display things. This keeps the markup simple. The general syntax for HTML tags is:

```
<tag parameters>normal text</tag>
```

As shown here, the tags usually come in pairs. The open tag may have some parameters, and the close tag name begins with a slash. The case of a tag is not considered, so `<title>`, `<Title>`, and `<TITLE>` are all valid and mean the same thing. The corresponding close tag could be `</title>`, `</Title>`, `</TITLE>`, or even `</TiTlE>`.

The `<A>` tag defines hypertext links that reference other pages on the Web. The hypertext links connect pages into a Web so that you can move from page to page to page and find related information. It is the flexibility of the links that make the Web so interesting. The `<A>` tag takes an `HREF` parameter that defines the destination of the link. If you wanted to link to my home page, you would put this in your page:

```
<A HREF="http://www.beedub.com/">Brent Welch</A>
```

When this construct appears in a Web page, your browser typically displays "Brent Welch" in blue underlined text. When you click on that text, your browser switches to the page at the address "http://www.beedub.com/". There is a lot more to HTML, of course, but this should give you a basic idea of what is going on in the examples. The following list summarizes the HTML tags that will be used in the examples:

Table 3–1 HTML tags used in the examples.

`HTML`	Main tag that surrounds the whole document.
`HEAD`	Delimits head section of the HTML document.
`TITLE`	Defines the title of the page.
`BODY`	Delimits the body section. Lets you specify page colors.

Table 3–1 HTML tags used in the examples. (Continued)

`H1 - H6`	HTML defines 6 heading levels: `H1`, `H2`, `H3`, `H4`, `H5`, `H6`.
`P`	Start a new paragraph.
`BR`	One blank line.
`B`	Bold text.
`I`	Italic text.
`A`	Used for hypertext links.
`IMG`	Specify an image.
`DL`	Definition list.
`DT`	Term clause in a definition list.
`DD`	Definition clause in a definition list.
`UL`	An unordered list.
`LI`	A bulleted item within a list.
`TABLE`	Create a table.
`TR`	A table row.
`TD`	A cell within a table row.
`FORM`	Defines a data entry form.
`INPUT`	A one-line entry field, checkbox, radio button, or submit button.
`TEXTAREA`	A multiline text field.

CGI for Dynamic Pages

There are two classes of pages on the Web, static and dynamic. A static page is written and stored on a Web server, and the same thing is returned each time a user views the page. This is the easy way to think about Web pages. You have some information to share, so you compose a page and tinker with the HTML tags to get the information to look good. If you have a home page, it is probably in this class.

In contrast, a dynamic page is computed each time it is viewed. This is how pages that give up-to-the-minute stock prices work, for example. A dynamic page does not mean it includes animations; it just means that a program computes the page contents when a user visits the page. The advantage of this approach is that a user might see something different each time he or she visits the page. As we shall see, it is also easier to maintain information in a database of some sort and generate the HTML formatting for the data with a program.

A CGI (Common Gateway Interface) program is used to compute Web pages. The CGI standard defines how inputs are passed to the program as well as a way to identify different types of results, such as images, plain text, or HTML markup. A CGI program simply writes the contents of the document to its standard output, and the Web server takes care of delivering the document to the user's Web browser. The following is a very simple CGI script:

Example 3–1 A simple CGI script.

```
puts "Content-Type: text/html"
puts ""
puts "<TITLE>The Current Time</TITLE>"
puts "The time is <B>[clock format [clock seconds]]</B>"
```

The program computes a simple HTML page that has the current time. Each time a user visits the page they will see the current time on the server. The server that has the CGI program and the user viewing the page might be on different sides of the planet. The output of the program starts with a `Content-Type` line that tells your Web browser what kind of data comes next. This is followed by a blank line and then the contents of the page.

The `clock` command is used twice: once to get the current time in seconds, and a second time to format the time into a nice looking string. The `clock` command is described in detail on page 173. Fortunately, there is no conflict between the markup syntax used by HTML and the Tcl syntax for embedded commands, so we can mix the two in the argument to the `puts` command. Double quotes are used to group the argument to `puts` so that the `clock` commands will be executed. When run, the output of the program will look like this:

Example 3–2 Output of Example 3–1.

```
Content-Type: text/html

<TITLE>The Current Time</TITLE>
The time is <B>Wed Oct 16 11:23:43  1996</B>
```

This example is a bit sloppy in its use of HTML, but it should display properly in most Web browsers. Example 3–3 includes all the required tags for a proper HTML document.

The `guestbook.cgi` Script

The `guestbook.cgi` script computes a page that lists all the registered guests. The example is shown first, and then each part of it is discussed in more detail later. One thing to note right away is that the HTML tags are generated by procedures that hide the details of the HTML syntax. The first lines of the script use the UNIX trick to have *tclsh* interpret the script. This trick is described on page 26:

Example 3–3 The guestbook.cgi script.

```
#!/bin/sh
# guestbook.cgi
# Implement a simple guestbook page.
# The set of visitors is kept in a simple database.
# The newguest.cgi script will update the database.
# \
exec tclsh "$0" ${1+"$@"}

# The cgilib.tcl file has helper procedures
# The guestbook.data file has the database
# Both file are in the same directory as the script

set dir [file dirname [info script]]
source [file join $dir cgilib.tcl]
set datafile [file join $dir guestbook.data]

Cgi_Header "Brent's Guestbook" {BGCOLOR=white TEXT=black}
P
if {![file exists $datafile]} {
   puts "No registered guests, yet."
   P
   puts "Be the first [Link {registered guest!} newguest.html]"
} else {
   puts "The following folks have registered in my GuestBook."
   P
   puts [Link Register newguest.html]
   H2 Guests
   catch {source $datafile}
   foreach name [lsort [array names Guestbook]] {
      set item $Guestbook($name)
      set homepage [lindex $item 0]
      set markup [lindex $item 1]
      H3 [Link $name $homepage]
      puts $markup
   }
}
Cgi_End
```

Using a Script Library File

The script uses a number of Tcl procedures that make working with HTML and the CGI interface easier. These procedures are kept in the cgilib.tcl file, which is kept in the same directory as the main script. The script starts by sourcing the cgilib.tcl file so that these procedures are available. The following command determines the location of the cgilib.tcl file based on the location of the main script. The info script command returns the file name of the script. The file dirname and file join commands manipulate file names in a platform-independent way. They are described on page 102. I use this trick to avoid putting absolute file names into my scripts, which would have to be changed if the program moves later:

```
set dir [file dirname [info script]]
source [file join $dir cgilib.tcl]
```

Beginning the HTML Page

The following command generates the standard information that comes at the beginning of an HTML page:

```
Cgi_Header {Brent’s GuestBook} {bgcolor=white text=black}
```

The `Cgi_Header` is shown in Example 3–4:

Example 3–4 The `Cgi_Header` procedure.

```
proc Cgi_Header {title {bodyparams {}}} {
    puts stdout \
"Content-Type: text/html

<HTML>
<HEAD>
<TITLE>$title</TITLE>
</HEAD>
<BODY $bodyparams>
<H1>$title</H1>"
}
```

The `Cgi_Header` procedure takes as arguments the title for the page and some optional parameters for the HTML `<Body>` tag. The guestbook.cgi script specifies black text on a white background to avoid the standard gray background of most browsers. The procedure definition uses the syntax for an optional parameter, so you do not have to pass `bodyparams` to `Cgi_Header`. Default values for procedure parameters are described on page 81.

The `Cgi_Header` procedure just contains a single `puts` command that generates the standard boilerplate that appears at the beginning of the output. Note that several lines are grouped together with double quotes. Double quotes are used so that the variable references mixed into the HTML are substituted properly.

The output begins with the CGI content-type information, a blank line, and then the HTML. The HTML is divided into a head and a body part. The `<TITLE>` tag goes in the head section of an HTML document. Finally, browsers display the title in a different place than the rest of the page, so I always want to repeat the title as a level-one heading (i.e., `H1`) in the body of the page.

Simple Tags and Hypertext Links

The next thing the program does is to see whether there are any registered guests or not. The `file` command, which is described in detail on page 102, is used to see whether there is any data:

```
if {![file exists $datafile]} {
```

If the database file does not exist, a different page is displayed to encourage a registration. The page includes a hypertext link to a registration page. The `newguest.html` page will be described in more detail later:

```
puts "No registered guests, yet."
P
puts "Be the first [Link {registered guest!} newguest.html]"
```

The `P` command generates the HTML for a paragraph break. This trivial procedure saves us a few keystrokes:

```
proc P {} {
   puts <P>
}
```

The `Link` command formats and returns the HTML for a hypertext link. Instead of printing the HTML directly, it is returned, so you can include it in-line with other text you are printing:

Example 3–5 The `Link` command formats a hypertext link.

```
proc Link {text url} {
    return "<A HREF=\"$url\">$text</A>"
}
```

The output of the program would be as below if there were no data:

Example 3–6 Initial output of `guestbook.cgi`.

```
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Brent’s Guestbook</TITLE>
</HEAD>
<BODY BGCOLOR=white TEXT=black>
<H1>Brent’s Guestbook</H1>
<P>
No registered guests.
<P>
Be the first <A HREF="newguest.html">registered guest!</A>
</BODY>
</HTML>
```

If the database file exists, then the real work begins. We first generate a link to the registration page, and a level-two header to separate that from the guest list:

```
puts [Link Register newguest.html]
H2 Guests
```

The H2 procedure handles the detail of including the matching close tag:

```
proc H2 {string} {
     puts "<H2>$string</H2>"
}
```

Using a Tcl Array for the Database

The datafile contains Tcl commands that define an array that holds the guestbook data. If this file is kept in the same directory as the guestbook.cgi script, then you can compute its name:

```
set dir [file dirname [info script]]
set datafile [file join $dir guestbook.data]
```

By using Tcl commands to represent the data, we can load the data with the source command. The catch command is used to protect the script from a bad data file, which will show up as an error from the source command. Catching errors is described in detail on page 79:

```
catch {source $datafile}
```

The Guestbook variable is the array defined in guestbook.data. Array variables are the topic of Chapter 8. Each element of the array is defined with a Tcl command that looks like this:

```
set Guestbook(key) {url markup}
```

The person's name is the array index, or key. The value of the array element is a Tcl list with two elements: their URL and some additional HTML markup that they can include in the guestbook. Tcl lists are the topic of Chapter 5. The following example shows what the command looks like with real data:

```
set {Guestbook(Brent Welch)} {
     http://www.beedub.com/
     {<img src=http://www.beedub.com/welch.gif>}
}
```

The spaces in the name result in additional braces to group the whole variable name and each list element. This syntax is explained on page 90. Do not worry about it now. We will see on page 42 that all the braces in the previous statement are generated automatically. The main point is that the person's name is the key, and the value is a list with two elements.

The array names command returns all the indices, or keys, in the array, and the lsort command sorts these alphabetically. The foreach command loops over the sorted list, setting the loop variable x to each key in turn:

```
foreach name [lsort [array names Guestbook]] {
```

Given the key, we get the value like this:

```
set item $Guestbook($name)
```

The two list elements are extracted with lindex, which is described on page 63.

```
set homepage [lindex $item 0]
set markup [lindex $item 1]
```

We generate the HTML for the guestbook entry as a level-three header that contains a hypertext link to the guest's home page. We follow the link with any HTML markup text that the guest has supplied to embellish his or her entry. The H3 procedure is similar to the H2 procedure already shown, except it generates <H3> tags:

```
H3 [Link $name $homepage]
puts $markup
```

Sample Output

The last thing the script does is call Cgi_End to output the proper closing tags. Example 3–7 shows the output of the guestbook.cgi script:

Example 3–7 Output of guestbook.cgi.

```
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Brent's Guestbook</TITLE>
</HEAD>
<BODY BGCOLOR=white TEXT=black>
<H1>Brent's Guestbook</H1>
<P>
The following folks have registered in my guestbook.
<P>
<A HREF="newguest.html">Register</A>
<H2>Guests</H2>
<H3><A HREF="http://www.beedub.com/">Brent Welch</A></H3>
<IMG SRC="http://www.beedub.com/welch.gif">
</BODY>
</HTML>
```

Defining Forms and Processing Form Data

The guestbook.cgi script only generates output. The other half of CGI deals with input from the user. Input is more complex for two reasons. First, we have to define another HTML page that has a form for the user to fill out. Second, the data from the form is organized and encoded in a standard form that must be decoded by the script. Example 3–8 on page 40 defines a very simple form, and the procedure that decodes the form data is shown in Example 11–6 on page 155.

The guestbook page contains a link to newguest.html. This page contains a form that lets a user register his or her name, home page URL, and some additional HTML markup. The form has a submit button. When a user clicks that button in their browser, the information from the form is passed to the newguest.cgi script. This script updates the database and computes another page for the user that acknowledges the user's contribution.

The `newguest.html` Form

An HTML form contains tags that define data entry fields, buttons, checkboxes, and other elements that let the user specify values. For example, a one-line entry field that is used to enter the home page URL is defined like this:

```
<INPUT TYPE=text NAME=url>
```

The `INPUT` tag is used to define several kinds of input elements, and its `type` parameter indicates what kind. In this case, `TYPE=text` creates a one-line text entry field. The submit button is defined with an `INPUT` tag that has `TYPE=submit`, and the `VALUE` parameter becomes the text that appears on the button:

```
<INPUT TYPE=submit NAME=submit VALUE=Register>
```

A general type-in window is defined with the `TEXTAREA` tag. This creates a multiline, scrolling text field that is useful for specifying lots of information, such as a free-form comment. In our case we will let guests type in HTML that will appear with their guestbook entry. The text between the open and close `TEXTAREA` tags is inserted into the type-in window when the page is first displayed.

```
<TEXTAREA NAME=markup ROWS=10 COLS=50>Hello.</TEXTAREA>
```

A common parameter to the form tags is `NAME=`*`something`*. This name identifies the data that will come back from the form. The tags also have parameters that affect their display, such as the label on the submit button and the size of the text area. Those details are not important for our example. The complete form is shown in Example 3–8:

Example 3–8 The `newguest.html` form.

```
<!Doctype HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
<TITLE>Register in my Guestbook</TITLE>
<!-- Author: bwelch -->
<META HTTP-Equiv=Editor Content="SunLabs WebTk 1.0beta 10/
11/96">
</HEAD>
<BODY>

<FORM ACTION="newguest.cgi" METHOD="POST">

<H1>Register in my Guestbook</H1>
<UL>
<LI>Name <INPUT TYPE="text" NAME="name" SIZE="40">
<LI>URL  <INPUT TYPE="text" NAME="url" SIZE="40">
<P>
If you don't have a home page, you can use an email URL like
"mailto:welch@acm.org"
<LI>Additional HTML to include after your link:
<BR>

<TEXTAREA NAME="html" COLS="60" ROWS="15">
</TEXTAREA>
<LI><INPUT TYPE="submit" NAME="new" VALUE="Add me to your
```

```
guestbook">
<LI><INPUT TYPE="submit" NAME="update" VALUE="Update my
guestbook entry">
</UL>
</FORM>

</BODY>
</HTML>
```

The `newguest.cgi` Script

When the user clicks the Submit button in their browser, the data from the form is passed to the program identified by the `Action` parameter of the form tag. That program takes the data, does something useful with it, and then returns a new page for the browser to display. In our case the `FORM` tag names `newguest.cgi` as the program to handle the data:

```
<FORM ACTION=newguest.cgi METHOD=POST>
```

The CGI specification defines how the data from the form is passed to the program. The data is encoded and organized so that the program can figure out the values the user specified for each form element. The encoding is handled rather nicely with some regular expression tricks that are done in `Cgi_Parse`. `Cgi_Parse` saves the form data, and `Cgi_Value` gets a form value in the script. These procedures are described in Example 11–6 on page 155. Example 3–9 starts out by calling `Cgi_Parse`:

Example 3–9 The `newguest.cgi` script.

```
#!/bin/sh
# \
exec tclsh "$0" ${1+"$@"}
# source cgilib.tcl from the same directory as newguest.cgi

set dir [file dirname [info script]]
source [file join $dir cgilib.tcl]
set datafile [file join $dir guestbook.data]

Cgi_Parse

# Open the datafile in append mode

if [catch {open $datafile a} out] {
   Cgi_Header "Guestbook Registration Error" \
      {BGCOLOR=black TEXT=red}
   P
   puts "Cannot open the data file"
   P
   puts $out;# the error message
   exit 0
}
```

```
# Append a Tcl set command that defines the guest's entry

puts $out ""
puts $out [list set Guestbook([Cgi_Value name]) \
   [list [Cgi_Value url] [Cgi_Value html]]]
close $out

# Return a page to the browser

Cgi_Header "Guestbook Registration Confirmed" \
   {BGCOLOR=white TEXT=black}

puts "
<DL>
<DT>Name
<DD>[Cgi_Value name]
<DT>URL
<DD>[Link [Cgi_Value url] [Cgi_Value url]]
</DL>
[Cgi_Value html]
"

Cgi_End
```

The main idea of the newguest.cgi script is that it saves the data to a file as a Tcl command that defines an element of the Guestbook array. This lets the guestbook.cgi script simply load the data by using the Tcl source command. This trick of storing data as a Tcl script saves us from the chore of defining a new file format and writing code to parse it. Instead, we can rely on the well-tuned Tcl implementation to do the hard work for us efficiently.

The script opens the datafile in append mode so that it can add a new record to the end. Opening files is described in detail on page 110. The script uses a catch command to guard against errors. If an error occurs, a page explaining the error is returned to the user. Working with files is one of the most common sources of errors (permission denied, disk full, file-not-found, and so on), so I always open the file inside a catch statement:

```
if [catch {open $datafile a} out] {
    # an error occurred
} else {
    # open was ok
}
```

In this command, the variable out gets the result of the open command, which is either a file descriptor or an error message. This style of using catch is described in detail in Example 6–14 on page 77.

The script writes the data as a Tcl set command. The list command is used to format the data properly:

```
puts $out [list set Guestbook([Cgi_Value name]) \
    [list [Cgi_Value url] [Cgi_Value html]]]
```

There are two lists. First the url and html values are formatted into one list. This list will be the value of the array element. Then, the whole Tcl command is formed as a list. In simplified form, the command is generated from this:

```
list set variable value
```

Using the list command ensures that the result will always be a valid Tcl command that sets the variable to the given value. The list command is described in more detail on page 61.

The cgi.tcl Package

The cgilib.tcl file included with this book just barely scratches the surface of things you might like to do in a CGI script. Don Libes has created a comprehensive package for CGI scripts known as cgi.tcl. You can find it on the Web at

```
http://expect.nist.gov/cgi.tcl/
```

One of Don's goals in cgi.tcl was to eliminate the need to directly write any HTML markup at all. Instead, he has defined a whole suite of Tcl commands similar to the P and H2 procedures shown in this chapter that automatically emit the matching close tags. He also has support procedures to deal with browser cookies, page redirects, and other CGI features.

Next Steps

There are a number of details that can be added to this example. A user may want to update their entry, for example. They could do that now, but they would have to retype everything. They might also like a chance to check the results of their registration and make changes before committing them. This requires another page that displays their guest entry as it would appear on a page, and also has the fields that let them update the data.

The details of how a CGI script is hooked up with a Web server vary from server to server. You should ask your local Webmaster for help if you want to try this out on your local Web site. The Tcl Web Server comes with this guestbook example already set up, plus it has a number of other very interesting ways to generate pages. My own taste in Web page generation has shifted from CGI to a template-based approach supported by the Tcl Web Server. This is the topic of Chapter 18.

The next few chapters describe basic Tcl commands and data structures. We return to the CGI example in Chapter 11 on regular expressions.

CHAPTER 4

String Processing in Tcl

This chapter describes string manipulation and simple pattern matching. Tcl commands described are: `string`, `append`, `format`, `scan`, and `binary`. The `string` command is a collection of several useful string manipulation operations.

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. This chapter describes a simple pattern matching mechanism that is similar to that used in many other shell languages. Chapter 11 describes a more complex and powerful regular expression pattern matching mechanism.

The `string` Command

The `string` command is really a collection of operations you can perform on strings. The following example calculates the length of the value of a variable.

```
set name "Brent Welch"
string length $name
=> 11
```

The first argument to `string` determines the operation. You can ask `string` for valid operations by giving it a bad one:

```
string junk
=> bad option "junk": should be bytelength, compare,
equal, first, index, is, last, length, map, match, range,
repeat, replace, tolower, totitle, toupper, trim, trim-
left, trimright, wordend, or wordstart
```

This trick of feeding a Tcl command bad arguments to find out its usage is common across many commands. Table 4–1 summarizes the `string` command.

Table 4–1 The `string` command.

`string bytelength` *`str`*	Returns the number of bytes used to store a string, which may be different from the character length returned by `string length` because of UTF-8 encoding. See page 210 of Chapter 15 about Unicode and UTF-8.

`string compare ?-nocase?`
`?-length` *`len`*`?` *`str1 str2`*
	Compares strings lexicographically. Use `-nocase` for case insensitve comparison. Use `-length` to limit the comparison to the first *`len`* characters. Returns 0 if equal, -1 if *`str1`* sorts before *`str2`*, else 1.

`string equal ?-nocase?`
`str1 str2`
	Compares strings and returns 1 if they are the same. Use `-nocase` for case insensitve comparison.

`string first` *`str1 str2`*	Returns the index in *`str2`* of the first occurrence of *`str1`*, or -1 if *`str1`* is not found.
`string index` *`string index`*	Returns the character at the specified *`index`*. An index counts from zero. Use `end` for the last character.

`string is` *`class`* `?-strict?`
`?-failindex` *`varname`*`?`
`string`
	Returns 1 if *`string`* belongs to *`class`*. If `-strict`, then empty strings never match, otherwise they always match. If `-failindex` is specified, then *`varname`* is assigned the index of the character in *`string`* that prevented it from being a member of *`class`*. See Table 4–3 on page 50 for character class names.

`string last` *`str1 str2`*	Returns the index in *`str2`* of the last occurrence of *`str1`*, or -1 if *`str1`* is not found.
`string length` *`string`*	Returns the number of characters in *`string`*.

`string map ?-nocase?`
`charMap string`
	Returns a new string created by mapping characters in *`string`* according to the input, output list in *`charMap`*. See page 51.

`string match` *`pattern str`*	Returns 1 if *`str`* matches the *`pattern`*, else 0. Glob-style matching is used. See page 48.
`string range` *`str i j`*	Returns the range of characters in *`str`* from *`i`* to *`j`*.
`string repeat` *`str count`*	Returns *`str`* repeated *`count`* times.

`string replace` *`str first`*
`last` `?`*`newstr`*`?`
	Returns a new string created by replacing characters *`first`* through *`last`* with *`newstr`*, or nothing.

`string tolower` *`string`*
`?`*`first`*`? ?`*`last`*`?`
	Returns *`string`* in lower case. *`first`* and *`last`* determine the range of *`string`* on which to operate.

`string totitle` *`string`*
`?`*`first`*`? ?`*`last`*`?`
	Capitalizes *`string`* by replacing its first character with the Unicode title case, or upper case, and the rest with lower case. *`first`* and *`last`* determine the range of *`string`* on which to operate.

Table 4–1 The `string` command. (Continued)

`string toupper` *`string`* `?`*`first`*`? ?`*`last`*`?`	Returns *`string`* in upper case. *`first`* and *`last`* determine the range of *`string`* on which to operate.
`string trim` *`string`* `?`*`chars`*`?`	Trims the characters in *`chars`* from both ends of *`string`*. *`chars`* defaults to whitespace.
`string trimleft` *`string`* `?`*`chars`*`?`	Trims the characters in *`chars`* from the beginning of *`string`*. *`chars`* defaults to whitespace.
`string trimright` *`string`* `?`*`chars`*`?`	Trims the characters in *`chars`* from the end of *`string`*. *`chars`* defaults to whitespace.
`string wordend` *`str ix`*	Returns the index in *`str`* of the character after the word containing the character at index *`ix`*.
`string wordstart` *`str ix`*	Returns the index in *`str`* of the first character in the word containing the character at index *`ix`*.

These are the string operations I use most:

- The `equal` operation, which is shown in Example 4–2 on page 48.
- String `match`. This pattern matching operation is described on page 48.
- The `tolower, totitle,` and `toupper` operations convert case.
- The `trim`, `trimright`, and `trimleft` operations are handy for cleaning up strings.

These new operations were added in Tcl 8.1 (actually, they first appeared in the 8.1.1 patch release):

- The `equal` operation, which is simpler than using `string compare`.
- The `is` operation that test for kinds of strings. String classes are listed in Table 4–3 on page 50.
- The `map` operation that translates characters (e.g., like the Unix *tr* command.)
- The `repeat` and `replace` operations.
- The `totitle` operation, which is handy for capitalizing words.

String Indices

Several of the string operations involve string indices that are positions within a string. Tcl counts characters in strings starting with zero. The special index `end` is used to specify the last character in a string:

```
string range abcd 2 end
=> cd
```

Tcl 8.1 added syntax for specifying an index relative to the end. Specify `end-`*`N`* to get the *`N`*th caracter before the end. For example, the following command returns a new string that drops the first and last characters from the original:

```
string range $string 1 end-1
```

There are several operations that pick apart strings: `first`, `last`, `wordstart`, `wordend`, `index`, and `range`. If you find yourself using combinations of these operations to pick apart data, it will be faster if you can do it with the regular expression pattern matcher described in Chapter 11.

Strings and Expressions

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

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

`expr` *is unreliable for string comparison.*

Ironically, despite the quotes, the expression evaluator first converts items to numbers if possible, and then converts them back if it detects a case of string comparison. The conversion back is always done as a decimal number. This can lead to unexpected conversions between strings that look like hexadecimal or octal numbers. The following boolean expression is true!

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

A safe way to compare strings is to use the `string compare` and `equal` operations. These operations work faster because the unnecessary conversions are eliminated. Like the C library `strcmp` function, `string compare` returns 0 if the strings are equal, minus 1 if the first string is lexicographically less than the second, or 1 if the first string is greater than the second:

Example 4–1 Comparing strings with `string compare`.

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

The `string equal` command added in Tcl 8.1 makes this simpler:

Example 4–2 Comparing strings with `string equal`.

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

String Matching

The `string match` command implements *glob*-style pattern matching that is modeled after the file name pattern matching done by various UNIX shells.

The heritage of the word "glob" is rooted in UNIX, and Tcl preserves this historical oddity in the glob command that does pattern matching on file names. The glob command is described on page 115. Table 4–2 shows the three constructs used in string match patterns:

Table 4–2 Matching characters used with string match.

*	Match any number of any characters.
?	Match exactly one character.
[*chars*]	Match any character in *chars*.

Any other characters in a pattern are taken as literals that must match the input exactly. The following example matches all strings that begin with a:

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

To match all two-letter strings:

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

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

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

Be careful! Square brackets are also special to the Tcl interpreter, so you will 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
```

You can specify 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. You can include more than one range in a set. Any letter, digit, or the underscore is matched with:

```
string match {[a-zA-Z0-9_]} $char
```

The set matches only a single character. To match more complicated patterns, like one or more characters from a set, then you need to use regular expression matching, which is described on page 148.

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 use two backslashes. They are replaced with a single backslash by Tcl before `string match` is called.

```
string match *\\? what?
```

Character Classes

The `string is` command tests a string to see whether it belongs to a particular *class*. This is useful for input validation. For example, to make sure something is a number, you do:

```
if {![string is integer $input]} {
    error "Invalid input. Please enter a number."
}
```

Classes are defined in terms of the Unicode character set, which means they are more general than specifying character sets with ranges over the ASCII encoding. For example, `alpha` includes many characters outside the range of `[A-Za-z]` because of different characters in other alphabets. The classes are listed in Table 4–3.

Table 4–3 Character class names.

`alnum`	Any alphabet or digit character.
`alpha`	Any alphabet character.
`ascii`	Any character with a 7-bit character code (i.e., less than 128.)
`boolean`	`0`, `1`, `true`, `false` (in any case).
`control`	Character code less than 32, and not NULL.
`digit`	Any digit character.
`double`	A valid floating point number.
`false`	`0` or `false` (in any case).
`graph`	Any printing characters, not including space characters.
`integer`	A valid integer.
`lower`	A string in all lower case.
`print`	A synonym for `alnum`.
`punct`	Any punctuation character.
`space`	Space, tab, newline, carriage return, vertical tab, backspace.
`true`	`1` or `true` (in any case).
`upper`	A string all in upper case.
`wordchar`	Alphabet, digit, and the underscore.
`xdigit`	Valid hexadecimal digits.

Mapping Strings

The `string map` command translates a string based on a character map. The map is in the form of a input, output list. Whereever a string contains an input sequence, that is replaced with the corresponding output. For example:

```
string map "food" {f p d l}
=> pool
```

The inputs and outputs can be more than one character and do not have to be the same length:

```
string map "food" {f p d ll oo u}
=> pull
```

Example 4–3 is more practical. It uses `string map` to replace fancy quotes and hyphens produced by Microsoft Word into ASCII equivalents. It uses the `open`, `read`, and `close` file operations that are described in Chapter 9, and the `fconfigure` command described on page 223 to ensure that the file format is UNIX friendly.

Example 4–3 Mapping Microsoft World special characters to ASCII.

```
proc Dos2Unix {filename} {
   set input [open $filename]
   set output [open $filename.new]
   fconfigure $output -translation lf
   puts $output [string map {
      \223   "
      \224   "
      \222   ’
      \226   -
   } [read $input]]
   close $input
   close $output
}
```

The append Command

The `append` command takes a variable name as its first argument and concatenates its remaining arguments onto the current value of the named variable. The variable is created if it does not already exist:

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

The append command is efficient with large strings.

The `append` command provides an efficient way to add items to the end of a string. It modifies a variable directly, so it can exploit the memory allocation scheme used internally by Tcl. Using the `append` command like this:

```
        append x " some new stuff"
```

is always faster than this:

```
        set x "$x some new stuff"
```

The lappend command described on page 61 has similar performance benefits when working with Tcl lists.

The format Command

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

```
        format spec value1 value2 ...
```

The spec argument includes literals and keywords. The literals are placed in the result as is, while each keyword indicates how to format the corresponding argument. The keywords are introduced with a percent sign, %, followed by zero or more modifiers, and terminate 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 most general keyword specification for each argument contains up to six parts:

- position specifier
- flags
- field width
- precision
- word length
- conversion character

These components are explained by a series of examples. The examples use double quotes around the format specification. This is 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. Table 4–4 lists the conversion characters:

Table 4–4 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 lowercase results.
c	Map from an integer to the ASCII character it represents.
s	A string.
f	Floating point number in the format a.b.

Table 4–4 Format conversions. (Continued)

`e or E`	Floating point number in scientific notation, `a.bE+-c`.
`g or G`	Floating point number in either `%f` or `%e` format, whichever is shorter.

A position specifier is *`i`*`$`, which means take the value from argument *`i`* as opposed to the normally corresponding argument. The position counts from 1. If a position is specified for one format keyword, the position must be used for all of them. If you group the format specification with double quotes, you need to quote the `$` with a backslash:

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

The position specifier is useful for picking a string from a set, such as this simple language-specific example. The message catalog facility described in Chapter 15 is a much more sophisticated way to solve this problem. The position is also useful if the same value is repeated in the formatted string.

The flags in a format are used to specify padding and justification. In the following examples, the `#` causes a leading `0x` to be printed in the hexadecimal value. The zero in `08` causes the field to be padded with zeros. Table 4–5 summarizes the format flag characters.

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

Table 4–5 Format flags.

`-`	Left justify the field.
`+`	Always include a sign, either + or -.
`space`	Precede 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).

After the flags you can specify a minimum field width value. The value is padded to this width with spaces, or 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 long words.

The scan Command

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

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

The format for `scan` is nearly the same as in the `format` command. There is no `%u` scan format. The `%c` scan format converts one character to its decimal value.

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

```
scan abcABC {%[a-z]} result
=> 1
set result
=> abc
```

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

The binary Command

Tcl 8.0 added support for binary strings. Previous versions of Tcl used null-terminated strings internally, which foils the manipulation of some types of data. Tcl now uses counted strings, so it can tolerate a null byte in a string value without truncating it.

This section describes the `binary` command that provides conversions between strings and packed binary data representations. The `binary format` command takes values and packs them according to a template. For example, this can be used to format a floating point vector in memory suitable for passing to Fortran. The resulting binary value is returned:

```
binary format template value ?value ...?
```

The `binary scan` command extracts values from a binary string according to a similar template. For example, this is useful for extracting data stored in binary format. It assigns values to a set of Tcl variables:

```
binary scan value template variable ?variable ...?
```

Format Templates

The template consists of type keys and counts. The types are summarized in Table 4–6. In the table, *count* is the optional count following the type letter.

Table 4–6 Binary conversion types.

`a`	A character string of length *count*. Padded with nulls in `binary format`.
`A`	A character string of length *count*. Padded with spaces in `binary format`. Trailing nulls and blanks are discarded in `binary scan`.
`b`	A binary string of length *count*. Low-to-high order.
`B`	A binary string of length *count*. High-to-low order.
`h`	A hexadecimal string of length *count*. Low-to-high order.
`H`	A hexadecimal string of length *count*. High-to-low order. (More commonly used than `h`.)
`c`	An 8-bit character code. The *count* is for repetition.
`s`	A 16-bit integer in little-endian byte order. The *count* is for repetition.
`S`	A 16-bit integer in big-endian byte order. The *count* is for repetition.
`i`	A 32-bit integer in little-endian byte order. The *count* is for repetition.
`I`	A 32-bit integer in big-endian byte order. The *count* is for repetition.
`f`	Single-precision floating point value in native format. *count* is for repetition.
`d`	Double-precision floating point value in native format. *count* is for repetition.
`x`	Pack *count* null bytes with `binary format`. Skip *count* bytes with `binary scan`.
`X`	Backup *count* bytes.
`@`	Skip to absolute position specified by *count*. If *count* is `*`, skip to the end.

The count is interpreted differently depending on the type. For types like integer (`i`) and double (`d`), the count is a repetition count (e.g., `i3` means three integers). For strings, the count is a length (e.g., `a3` means a three-character string). If no count is specified, it defaults to 1. If count is `*`, then `binary scan` uses all the remaining bytes in the value.

Several type keys can be specified in a template. Each key-count combination moves an imaginary cursor through the binary data. There are special type keys to move the cursor. The `x` key generates null bytes in `binary format,` and it skips over bytes in `binary scan`. The `@` key uses its *`count`* as an absolute byte offset to which to set the cursor. As a special case, `@*` skips to the end of the data. The `X` key backs up *`count`* bytes.

Numeric types have a particular byte order that determines how their value is laid out in memory. The type keys are lowercase for little-endian byte order (e.g., Intel) and uppercase for big-endian byte order (e.g., SPARC and Motorola). Different integer sizes are 16-bit (`s` or `S`), 32-bit (`i` or `I`), and possibly 64-bit (`l` or `L`) on those machines that support 64-bit integers. Note that the official byte order for data transmitted over a network is big-endian. Floating point values are always machine-specific, so it only makes sense to format and scan these values on the same machine.

There are three string types: character (`a` or `A`), binary (`b` or `B`), and hexadecimal (`h` or `H`). With these types the *`count`* is the length of the string. The `a` type pads its value to the specified length with null bytes in `binary format` and the `A` type pads its value with spaces. If the value is too long, it is truncated. In `binary scan`, the `A` type strips trailing blanks and nulls.

A binary string consists of zeros and ones. The `b` type specifies bits from low-to-high order, and the `B` type specifies bits from high-to-low order. A hexadecimal string specifies 4 bits (i.e., nybbles) with each character. The `h` type specifies nybbles from low-to-high order, and the `H` type specifies nybbles from high-to-low order. The `B` and `H` formats match the way you normally write out numbers.

Examples

When you experiment with `binary format` and `binary scan`, remember that Tcl treats things as strings by default. A "6", for example, is the character 6 with character code 54 or 0x36. The `c` type returns these character codes:

```
set input 6
binary scan $input "c" 6val
set 6val
=> 54
```

You can scan several character codes at a time:

```
binary scan abc "c3" list
=> 1
set list
=> 97 98 99
```

The previous example uses a single type key, so `binary scan` sets one corresponding Tcl variable. If you want each character code in a separate variable, use separate type keys:

```
binary scan abc "ccc" x y z
=> 3
set z
=> 99
```

Use the `H` format to get hexadecimal values:

```
binary scan 6 "H2" 6val
set 6val
=> 36
```

Use the `a` and `A` formats to extract fixed width fields. Here the `*` count is used to get all the rest of the string. Note that `A` trims trailing spaces:

```
binary scan "hello world " a3x2A* first second
puts "\"$first\" \"$second\""
=> "hel" " world"
```

Use the `@` key to seek to a particular offset in a value. The following command gets the second double-precision number from a vector. Assume the vector is read from a binary data file:

```
binary scan $vector "@8d" double
```

With `binary format`, the `a` and `A` types create fixed width fields. `A` pads its field with spaces, if necessary. The value is truncated if the string is too long:

```
binary format "A9A3" hello world
=> hello    wor
```

An array of floating point values can be created with this command:

```
binary format "f*" 1.2 3.45 7.43 -45.67 1.03e4
```

Remember that floating point values are always in native format, so you have to read them on the same type of machine that they were created. With integer data you specify either big-endian or little-endian formats. The `tcl_platform` variable described on page 182 can tell you the byte order of the current platform.

Binary Data and File I/O

When working with binary data in files, you need to turn off the newline translations and character set encoding that Tcl performs automatically. These are described in more detail on pages 114 and 209. For example, if you are generating binary data, the following command puts your standard output in binary mode:

```
fconfigure stdout -translation binary -encoding binary
puts [binary format "B8" 11001010]
```

Related Chapters

- To learn more about manipulating data in Tcl, read about lists in Chapter 5 and arrays in Chapter 8.
- For more about pattern matching, read about regular expressions in Chapter 11.
- For more about file I/O, see Chapter 9.
- For information on Unicode and other Internationalization issues, see Chapter 15.

CHAPTER 5

Tcl Lists

This chapter describes Tcl lists. Tcl commands described are: `list`, `lindex`, `llength`, `lrange`, `lappend`, `linsert`, `lreplace`, `lsearch`, `lsort`, `concat`, `join`, and `split`.

Lists in Tcl have the same structure as Tcl commands. All the rules you learned about grouping arguments in Chapter 1 apply to creating valid Tcl lists. However, when you work with Tcl lists, it is best to think of lists in terms of operations instead of syntax. Tcl commands provide operations to put values into a list, get elements from lists, count the elements of lists, replace elements of lists, and so on. The syntax can sometimes be confusing, especially when you have to group arguments to the list commands themselves.

Lists are used with commands such as `foreach` that take lists as arguments. In addition, lists are important when you are building up a command to be evaluated later. Delayed command evaluation with `eval` is described in Chapter 10, and similar issues with Tk callback commands are described in Chapter 27.

However, Tcl lists are not often the right way to build complicated data structures in scripts. You may find Tcl arrays more useful, and they are the topic of Chapter 8. List operations are also not right for handling unstructured data such as user input. Use regular expressions instead, which are described in Chapter 11.

Tcl Lists

A Tcl list is a sequence of values. When you write out a list, it has the same syntax as a Tcl command. A list has its elements separated by white space. Braces or quotes can be used to group words with white space into a single list element.

Because of the relationship between lists and commands, the list-related commands described in this chapter are used often when constructing Tcl commands.

Big lists were often slow before Tcl 8.0.

Unlike list data structures in other languages, Tcl lists are just strings with a special interpretation. The string representation must be parsed on each list access, so be careful when you use large lists. A list with a few elements will not slow down your code much. A list with hundreds or thousands of elements can be very slow. If you find yourself maintaining large lists that must be frequently accessed, consider changing your code to use arrays instead.

The performance of lists was improved by the Tcl compiler added in Tcl 8.0. The compiler stores lists in an internal format that requires constant time to access. Accessing the first element costs the same as accessing any other element in the list. Before Tcl 8.0, the cost of accessing an element was proportional to the number of elements before it in the list. The internal format also records the number of list elements, so getting the length of a list is cheap. Before Tcl 8.0, computing the length required reading the whole list.

Table 5–1 briefly describes the Tcl commands related to lists.

Table 5–1 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`* ...	Appends elements to the value of *`listVar`*.
`linsert` *`list index arg arg`* ...	Inserts elements into *`list`* before the element at position *`index`*. Returns a new list.
`lreplace` *`list i j arg arg`* `...`	Replaces elements *`i`* through *`j`* of *`list`* with the *`args`*. Returns a new list.
`lsearch ?`*`mode`*`?` *`list value`*	Returns 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. Returns -1 if not found.
`lsort ?`*`switches`*`?` *`list`*	Sorts elements of the list according to the switches: `-ascii`, `-integer`, `-real`, `-dictionary`, `-increasing`, `-decreasing`, `-index` *`ix`*, `-command` *`command`*. Returns a new list.
`concat` *`list list`* `...`	Joins multiple lists together into one list.
`join` *`list joinString`*	Merges the elements of a list together by separating them with *`joinString`*.
`split` *`string splitChars`*	Splits a string up into list elements, using the characters in *`splitChars`* as boundaries between list elements.

Constructing Lists

Constructing a list can be tricky because you must maintain proper list syntax. In simple cases, you can do this by hand. In more complex cases, however, you should use Tcl commands that take care of quoting so that the syntax comes out right.

The `list` command

The `list` command constructs a list out of its arguments so that there is one list element for each argument. If any of the arguments contain special characters, the `list` command adds quoting to ensure that they are parsed as a single element of the resulting list. The automatic quoting is very useful, and the examples in this book use the `list` command frequently. The next example uses `list` to create a list with three values, two of which contain special characters.

Example 5–1 Constructing a list with the `list` command.

```
set x {1 2}
=> 1 2
set y foo
=> foo
set l1 [list $x "a b" $y]
=> {1 2} {a b} foo
set l2 "\{$x\} {a b} $y"
=> {1 2} {a b} foo
```

The `list` command does automatic quoting.

Compare the use of `list` with doing the quoting by hand in Example 5–1. The assignment of `l2` requires carefully constructing the first list element by using quoted braces. The braces must be turned off so that `$x` can be substituted, but we need to group the result so that it remains a single list element. We also have to know in advance that `$x` contains a space, so quoting is required. We are taking a risk by not quoting `$y` because we know it doesn't contain spaces. If its value changes in the future, the structure of the list can change and even become invalid. In contrast, the `list` command takes care of all these details automatically.

When I first experimented with Tcl lists, I became confused by the treatment of curly braces. In the assignment to `x`, for example, the curly braces disappear. However, they come back again when `$x` is put into a bigger list. Also, the double quotes around `a b` get changed into curly braces. What's going on? Remember that there are two steps. In the first step, the Tcl parser groups arguments. In the grouping process, the braces and quotes are syntax that define groups. These syntax characters get stripped off. The braces and quotes are not part of the value. In the second step, the `list` command creates a valid Tcl list. This may require quoting to get the list elements into the right groups. The `list` command uses curly braces to group values back into list elements.

The `lappend` Command

The `lappend` command is used to append elements to the end of a list. The first argument to `lappend` is the name of a Tcl variable, and the rest of the arguments are added to the variable's value as new list elements. Like `list`, `lappend` preserves the structure of its arguments. It may add braces to group the values of its arguments so that they retain their identity as list elements when they are appended onto the string representation of the list.

Example 5–2 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 `lappend` command is implemented efficiently to take advantage of the way that Tcl stores lists internally. It is always more efficient to use `lappend` than to try and append elements by hand.

The `concat` Command

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

Example 5–3 Using `concat` to splice lists together.

```
set x {4 5 6}
set y {2 3}
set z 1
concat $z $y $x
=> 1 2 3 4 5 6
```

Double quotes behave much like the `concat` command. In simple cases, double quotes behave exactly like `concat`. However, the `concat` command trims extra white space from the end of its arguments before joining them together with a single separating space character. Example 5–4 compares the use of `list`, `concat`, and double quotes:

Example 5–4 Double quotes compared to the concat and list commands.

```
set x {1 2}
=> 1 2
set y "$x 3"
=> 1 2 3
set y [concat $x 3]
=> 1 2 3
set s { 2 }
=>  2
set y "1 $s 3"
=> 1  2  3
set y [concat 1 $s 3]
=> 1 2 3
set z [list $x $s 3]
=> {1 2} { 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) eliminates 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 10.

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
llength {}
=> 0
```

The lindex command returns a particular element of a list. It takes an index; list indices count from zero.

```
set x {1 2 3}
lindex $x 1
=> 2
```

You can use the keyword end to specify the last element of a list, or the syntax end-*N* to count back from the end of the list. The following commands are equivalent ways to get the element just before the last element in a list.

```
lindex $list [expr {[llength $list] - 2}]
lindex $list end-1
```

The lrange command returns a range of list elements. It takes a list and two indices as arguments. Again, end or end-*N* can be used as an index:

```
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 zero or less, then the elements are added to the front. If the index is equal to or greater than the length of the list, then the elements are appended to the end. Otherwise, the elements are inserted before the element that is currently at the specified index.

lreplace replaces a range of list elements with new elements. If you don't specify any new elements, you effectively delete elements from a list.

Note: linsert and lreplace do not modify an existing list. Instead, they return a new list value. In the following example, the lreplace command does not change the value of x:

Example 5–5 Modifying lists with linsert and lreplace.

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

Searching Lists: lsearch

lsearch returns the index of a value in the list, or -1 if it is not present. lsearch supports pattern matching in its search. Glob-style pattern matching is the default, and this can be disabled with the -exact flag. The semantics of glob pattern matching is described in Chapter 4. The -regexp option lets you specify the list value with a regular expression. Regular expressions are described in Chapter 11. In the following example, the glob pattern l* matches the value list.

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

Example 5–6 uses lreplace and lsearch to delete a list element by value. The value is found with lsearch. The value is removed with an lreplace that does not specify any replacement list elements:

Example 5–6 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 list is not sorted in place. Instead, a new list value is returned. The basic types of sorts are specified with the `-ascii`, `-dictionary`, `-integer`, or `-real` options. The `-increasing` or `-decreasing` option indicate the sorting order. The default option set is `-ascii -increasing`. An ASCII sort uses character codes, and a dictionary sort folds together case and treats digits like numbers. For example:

```
lsort -ascii {a Z n2 n100}
=> Z a n100 n2
lsort -dictionary {a Z n2 n100}
=> a n2 n100 Z
```

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

Example 5–7 Sorting a list using a comparison function.

```
proc NameCompare {a b} {
   set alast [lindex $a end]
   set blast [lindex $b end]
   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 NameCompare $list
=> {Miles Davis} {John Ousterhout} {Brent B. Welch}
```

The `NameCompare` 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.

Tcl 8.0 added a `-index` option to `lsort` that can be used to sort lists on an index. Instead of using `NameCompare`, you could do this:

```
lsort -index end $list
```

The `split` Command

The `split` command takes a string and turns it into a list by breaking it at specified characters and ensuring that the result has the proper list syntax. The `split` command provides a robust way to turn input lines into proper Tcl lists:

```
set line {welch:*:28405:100:Brent Welch:/usr/welch:/bin/csh}
split $line :
=> welch * 28405 100 {Brent Welch} /usr/welch /bin/csh
lindex [split $line :] 4
=> Brent Welch
```

Do not use list operations on arbitrary data.

Even if your data has space-separated words, you should be 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. Your code will work with simple test cases, but when invalid list syntax appears in the input, your script will raise an error. The next example shows what happens when input is not a valid list. The syntax error, an unmatched quote, occurs in the middle of the list. However, you cannot access any of the list because the `lindex` command tries to convert the value to a list before returning any part of it.

Example 5–8 Use `split` to turn input data into Tcl lists.

```
set line {this is "not a tcl list}
lindex $line 1
=> unmatched open quote in list
lindex [split $line] 2
=> "not
```

The default separator character for `split` is white space, which contains spaces, tabs, and newlines. 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. The backslash–space sequence is used to include a space in the set of characters. You could also group the argument to `split` with double quotes:

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

A trick that splits each character into a list element is to specify an empty string as the split character. This lets you get at individual characters with list operations:

```
split abc {}
=> a b c
```

However, if you write scripts that process data one character at a time, they may run slowly. Read Chapter 11 about regular expressions for hints on really efficient string processing.

The join Command

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 removes 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
```

If the treatment of braces is puzzling, remember that the first value is parsed into a list. The braces around element values disappear in the process. Example 5–9 shows a way to implement join in a Tcl procedure, which may help to understand the process:

Example 5–9 Implementing join in Tcl.

```
proc join {list sep} {
   set s {}  ;# s is the current separator
   set result {}
   foreach x $list {
      append result $s $x
      set s $sep
   }
   return $result
}
```

Related Chapters

- Arrays are the other main data structure in Tcl. They are described in Chapter 8.
- List operations are used when generating Tcl code dynamically. Chapter 10 describes these techniques when using the eval command.
- The foreach command loops over the values in a list. It is described on page 73 in Chapter 6.

CHAPTER 6

Control Structure Commands

This chapter describes the Tcl commands that implement control structures: `if`, `switch`, `foreach`, `while`, `for`, `break`, `continue`, `catch`, `error`, and `return`.

Control structure 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 structures: `break`, `continue`, `return`, and `error`.

A control structure 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 structure command is invoked. Group with braces, and let the control structure command trigger evaluation at the proper time. A control structure 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 control structure 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 expression ?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 6–1 A conditional `if then else` command.

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

Curly brace positioning is important.

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 lines 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 performs variable and command substitution on the expression. Using curly braces ensures that these substitutions are performed at the proper time. It is possible to be lax in this regard, with constructs such as:

```
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`. This style is fragile and error prone. Instead, always use braces around the command bodies to avoid trouble later when you modify the command. The following is much better (use `then` if it suits your taste):

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

When you are testing the result of a command, you can get away without using curly braces around the command, like this:

```
if [command] body1
```

However, it turns out that you can execute the `if` statement more efficiently if you always group the expression with braces, like this:

```
if {[command]} body1
```

You can create chained conditionals by using the `elseif` keyword. Again, note the careful placement of curly braces that create a single `if` command:

Example 6–2 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. 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 4 and Chapter 11. The general form of the command is:

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

Any number of pattern-body pairs can be specified. If multiple patterns match, only the body of the first matching pattern is evaluated. You can also group all the pattern-body pairs into one argument:

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

The first form allows substitutions on the patterns but will require backslashes to continue the command onto multiple lines. This is shown in Example 6–4 on page 72. The second form groups all the patterns and bodies into one argument. This makes it easy to group the whole command without worrying about newlines, but it suppresses any substitutions on the patterns. This is shown in Example 6–3. In either case, you should always group the command bodies with curly braces so that substitution occurs only on the body with the pattern that matches the value.

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

`-exact` Matches the *`value`* exactly to one of the patterns. This is the default.

`-glob` Uses glob-style pattern matching. See page 48.

`-regexp` Uses regular expression pattern matching. See page 134.

`--` No flag (or end of flags). Necessary when *`value`* can begin with `-`.

The `switch` command raises an error if any other flag is specified or if the *`value`* begins with `-`. In practice I always use the `--` flag before *`value`* so that I don't have to worry about that problem.

If the pattern associated with the last body is `default`, then this command body is executed if no other patterns match. The `default` keyword works only 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`:

Example 6–3 Using `switch` for an exact match.

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

If you have variable references or backslash sequences in the patterns, then you cannot use braces around all the pattern-body pairs. You must use backslashes to escape the newlines in the command:

Example 6–4 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 Chapter 11.)

If the body associated with a pattern is just a dash, `-`, then the `switch` command "falls through" to the body associated with the next pattern. You can tie together any number of patterns in this manner.

Example 6–5 A `switch` with "fall through" cases.

```
switch -glob -- $value {
   X* -
   Y* { takeXorYaction $value }
}
```

Comments in `switch` Commands

A comment can occur only where the Tcl parser expects a command to begin. This restricts the location of comments in a `switch` command. You must put them inside the command body associated with a pattern, as shown in Example 6–6. If you put a comment at the same level as the patterns, the `switch` command will try to interpret the comment as one or more pattern-body pairs.

Example 6–6 Comments in `switch` commands.

```
switch -- $value {
   # this comment confuses switch
   pattern { # this comment is ok }
}
```

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 (nonzero). 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 1–13 on page 12):

```
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 6–7 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]
}
```

Foreach

The `foreach` command loops over a command body assigning one or more loop variables to each of the values in one or more lists. Multiple loop variables were introduced in Tcl 7.5. The syntax for the simple case of a single variable and a single list 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 list with the loop variable taking on successive values in the list. The list can be entered explicitly, as in the next example:

Example 6–8 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
```

It is also common to use a list-valued variable or command result instead of a static list value. The next example loops through command-line arguments. The variable `argv` is set by the Tcl interpreter to be a list of the command-line arguments given when the interpreter was started:

Example 6–9 Parsing command-line arguments.

```
# argv is set by the Tcl shells
# possible flags are:
# -max integer
# -force
# -verbose
set state flag
set force 0
set verbose 0
set max 10
foreach arg $argv {
   switch -- $state {
      flag {
         switch -glob -- $arg {
            -f*    {set force 1}
            -v*    {set verbose 1}
            -max   {set state max}
            default {error "unknown flag $arg"}
         }
      }
      max {
         set max $arg
         set state flag
      }
   }
}
```

The loop uses the `state` variable to keep track of what is expected next, which in this example is either a flag or the integer value for `-max`. The `--` flag to `switch` is *required* in this example because the `switch` command complains about a bad flag if the pattern begins with a `-` character. The `-glob` option lets the user abbreviate the `-force` and `-verbose` options.

If the list of values is to contain variable values or command results, then the `list` command should be used to form the list. Avoid double quotes 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 6–10 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.

Multiple Loop Variables

You can have more than one loop variable with `foreach`. Suppose you have two loop variables `x` and `y`. In the first iteration of the loop, `x` gets the first value from the value list and `y` gets the second value. In the second iteration, `x` gets the third value and `y` gets the fourth value. This continues until there are no more values. If there are not enough values to assign to all the loop variables, the extra variables get the empty string as their value.

Example 6–11 Multiple loop variables with `foreach`.

```
foreach {key value} {orange 55 blue 72 red 24 green} {
   puts "$key: $value"
}
orange: 55
blue: 72
red: 24
green:
```

If you have a command that returns a short list of values, then you can abuse the `foreach` command to assign the results of the commands to several variables all at once. For example, suppose the command `MinMax` returns two values as a list: the minimum and maximum values. Here is one way to get the values:

```
set result [MinMax $list]
set min [lindex $result 0]
set max [lindex $result 1]
```

The `foreach` command lets us do this much more compactly:

```
foreach {min max} [MinMax $list] {break}
```

The `break` in the body of the `foreach` loop guards against the case where the command returns more values than we expected. This trick is encapsulated into the `lassign` procedure in Example 10–4 on page 131.

Multiple Value Lists

The `foreach` command has the ability to loop over multiple value lists in parallel. In this case, each value list can also have one or more variables. The `foreach` command keeps iterating until all values are used from all value lists. If a value list runs out of values before the last iteration of the loop, its corresponding loop variables just get the empty string for their value.

Example 6–12 Multiple value lists with `foreach`.

```
foreach {k1 k2} {orange blue red green black} value {55 72 24} {
   puts "$k1 $k2: $value"
}
orange blue: 55
red green: 72
black : 24
```

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 whether the loop body will execute. The third argument is a command to execute after the loop body:

Example 6–13 A `for` loop.

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

You could use `for` to iterate over a list, but you should really use `foreach` instead. Code like the following is slow and cluttered:

```
for {set i 0} {$i < [llength $list]} {incr i} {
    set value [lindex $list $i]
}
```

This is the same as:

```
foreach value $list {
}
```

Break and Continue

You can control loop execution 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. There is no goto command in Tcl.

Catch

Until now we have ignored the possibility of errors. In practice, however, a command will raise an error if it is called with the wrong number of arguments, or if it detects some error condition particular to its implementation. An uncaught error aborts 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 zero if there was no error caught, or a nonzero error code if it did catch an error.

You should use curly braces to group the command instead of double quotes because catch invokes the full Tcl interpreter on the command. If double quotes are used, an extra round of substitutions occurs 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 6–14 A standard catch phrase.

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

A more 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:

* More precisely, the Tcl script unwinds and the current Tcl_Eval procedure in the C runtime library returns TCL_ERROR. There are three cases. In interactive use, the Tcl shell prints the error message. In Tk, errors that arise during event handling trigger a call to bgerror, a Tcl procedure you can implement in your application. In your own C code, you should check the result of Tcl_Eval and take appropriate action in the case of an error.

Example 6–15 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 acceptable because `catch` always returns an integer, 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.

Catching More Than Errors

The `catch` command catches more than just errors. If the command body contains `return`, `break`, or `continue` commands, these terminate the command body and are reflected by `catch` as nonzero return codes. You need to be aware of this if you try to isolate troublesome code with a `catch` phrase. An innocent looking `return` command will cause the `catch` to signal an apparent error. The next example uses `switch` to find out exactly what `catch` returns. Nonerror cases are passed up to the surrounding code by invoking `return`, `break`, or `continue`:

Example 6–16 There are several possible return values from `catch`.

```
switch [catch {
   command1
   command2
   ...
} result] {
   0 {                  # Normal completion }
   1 {                  # Error case }
   2 { return $result  ;# return from procedure}
   3 { break           ;# break out of the loop}
   4 { continue        ;# continue loop}
   default {            # User-defined error codes }
}
```

Error

The `error` command raises an error condition that terminates 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 6–17 Raising an error.

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

In the previous example, 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`. In the next example, the information from the original error is preserved:

Example 6–18 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 specifies a concise, machine-readable description of the error. It is stored into the global `errorCode` variable. It defaults to `NONE`. Many of the file system commands return an `errorCode` that has three elements: `POSIX`, the error name (e.g., `ENOENT`), and the associated error message:

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

In addition, your application can define error codes of its own. Catch phrases can 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. The -errorcode option sets the global errorCode variable, and the -errorinfo option initializes the errorInfo global variable. When you use return -code error, there is no error command in the stack trace. Compare Example 6–17 with Example 6–19:

Example 6–19 Raising an error with return.

```
proc bar {} {
   return -code error bogus
}
catch {bar} result
=> 1
set result
=> bogus
set errorInfo
=> bogus
   while executing
"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. These -code options to return enable the construction of new control structures entirely in Tcl. The following example implements the break command with a Tcl procedure:

```
proc break {} {
     return -code break
}
```

CHAPTER 7

Procedures and Scope

Procedures encapsulate a set of commands, and they introduce a local scope for variables. Commands described are: `proc`, `global`, and `upvar`.

Procedures 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. Originally, Tcl had one global scope for shared variables, local scopes within procedures, and one global scope for procedures. Tcl 8.0 added *namespaces* that provide new scopes for procedures and global variables. For simple applications you can ignore namespaces and just use the global scope. Namespaces are described in Chapter 14.

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 is added to the set of commands understood by the Tcl interpreter. The name is case sensitive and can contain any characters. Procedure names do not conflict with variable names. 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.

Procedures can have default parameters so that the caller can leave out some of the command arguments. A default parameter is specified with its name and default value, as shown in the next example:

Example 7–1 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 take on the values specified in the proc command. If two arguments are provided, then only c gets the default value, and the arguments are 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 7–2 Variable number of arguments.

```
proc ArgTest {a {b foo} args} {
   foreach param {a b args} {
      puts stdout "\t$param = [set $param]"
   }
}
set x one
set y {two things}
set z \[special\$
ArgTest $x
=> a = one
   b = foo
   args =
ArgTest $y $z
=> a = two things
   b = [special$
   args =
ArgTest $x $y $z
=> a = one
   b = two things
   args = {[special$}
ArgTest $z $y $z $x
=> a = [special$
   b = two things
   args = {[special$} one
```

The effect of the list structure in args is illustrated by the treatment of variable z in Example 7–2. The value of z has special characters in it. When $z is passed as the value of parameter b, its value comes through to the procedure unchanged. When $z is part of the optional parameters, quoting is automatically added to create a valid Tcl list as the value of args. Example 10–3 on page 127 illustrates a technique that uses eval to undo the effect of the added list structure.

Changing Command Names with rename

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

```
rename foo foo.orig
```

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

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

```
rename exec {}
```

Scope

By default there is a single, global scope for procedure names. This means that you can use a procedure anywhere in your script. Variables defined outside any procedure are global variables. However, as described below, global variables are not automatically visible inside procedures. There is a different namespace for variables and procedures, so you could have a procedure and a global variable with the same name without conflict. You can use the namespace facility described in Chapter 7 to manage procedures and global variables.

Each procedure has a local scope for variables. That is, variables introduced in the procedure live only 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. You can also use qualified names to name variables in a namespace scope. The global and upvar commands are described later in this chapter. Qualified names are described on page 198. If the same variable name exists in an outer scope, it is unaffected by the use of that variable name inside a procedure.

In Example 7–3, the variable a in the global scope is different from the parameter a to P1. Similarly, the global variable b is different from the variable b inside P1:

Example 7–3 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
```

The `global` Command

Global scope is the toplevel scope. This scope is outside of any procedure. Variables defined at the global scope must be made accessible to the commands inside a procedure by using the `global` command. The syntax for `global` is:

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

The `global` command goes inside a procedure.

The `global` command adds a global variable to the current scope. A common mistake is to have a single `global` command and expect that to apply to all procedures. However, a `global` command in the global scope has no effect. Instead, you must put a `global` command in all procedures that access the global variable. The variable can be undefined at the time the `global` command is used. When the variable is defined, it becomes visible in the global scope.

Example 7–4 shows a random number generator. Before we look at the example, let me point out that the best way to get random numbers in Tcl is to use the `rand()` math function:

```
expr rand()
=> .137287362934
```

The point of the example is to show a state variable, the seed, that has to persist between calls to `random`, so it is kept in a global variable. The choice of `randomSeed` as the name of the global variable associates it with the random number generator. It is important to pick names of global variables carefully to avoid conflict with other parts of your program. For comparison, Example 14–1 on page 196 uses namespaces to hide the state variable:

Example 7–4 A random number generator.[*]

```
proc RandomInit { seed } {
   global randomSeed
   set randomSeed $seed
}
proc Random {} {
   global randomSeed
   set randomSeed [expr ($randomSeed*9301 + 49297) % 233280]
   return [expr $randomSeed/double(233280)]
}
proc RandomRange { range } {
   expr int([Random]*$range)
}
RandomInit [pid]
=> 5049
Random
=> 0.517686899863
Random
=> 0.217176783265
RandomRange 100
=> 17
```

Call by Name Using `upvar`

Use the `upvar` command when you need to pass the name of a variable, as opposed to its value, into a procedure. 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 treated like an array. The use of `upvar` and arrays is discussed further in Chapter 8 on page 92. The following procedure uses `upvar` to print the value of a variable given its name.

[*] Adapted from *Exploring Expect* by Don Libes, O’Reilly & Associates, Inc., 1995, and from *Numerical Recipes in C* by Press et al., Cambridge University Press, 1988.

Example 7–5 Print variable by name.

```
proc PrintByName { varName } {
    upvar 1 $varName var
    puts stdout "$varName = $var"
}
```

You can use upvar to fix the incr command. One drawback of the built-in incr is that it raises an error if the variable does not exist. We can define a new version of incr that initializes the variable if it does not already exist:

Example 7–6 Improved incr procedure.

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

Variable Aliases with upvar

The upvar command is useful in any situation where you have the name of a variable stored in another variable. In Example 7–2 on page 82, the loop variable param holds the names of other variables. Their value is obtained with this construct:

```
puts stdout "\t$param = [set $param]"
```

Another way to do this is to use upvar. It eliminates the need to use awkward constructs like [set $param]. If the variable is in the same scope, use zero as the scope number with upvar. The following is equivalent:

```
upvar 0 $param x
puts stdout "\t$param = $x"
```

Associating State with Data

Suppose you have a program that maintains state about a set of objects like files, URLs, or people. You can use the name of these objects as the name of a variable that keeps state about the object. The upvar command makes this more convenient:

```
upvar #0 $name state
```

Using the name directly like this is somewhat risky. If there were an object named x, then this trick might conflict with an unrelated variable named x elsewhere in your program. You can modify the name to make this trick more robust:

```
upvar #0 state$name state
```

Your code can pass *name* around as a handle on an object, then use upvar to get access to the data associated with the object. Your code is just written to use the state variable, which is an alias to the state variable for the current object. This technique is illustrated in Example 17–7 on page 232.

Namespaces and upvar

You can use upvar to create aliases for namespace variables, too. Namespaces are described in Chapter 14. For example, as an alternative to reserving all global variables beginning with state, you can use a namespace to hide these variables:

```
upvar #0 state::$name state
```

Now state is an alias to the namespace variable. This upvar trick works from inside any namespace.

Commands That Take Variable Names

Several Tcl commands involve variable names. For example, the Tk widgets can be associated with a global Tcl variable. The vwait and tkwait commands also take variable names as arguments.

Upvar aliases do not work with text variables.

The aliases created with upvar do not work with these commands, nor do they work if you use trace, which is described on page 183. Instead, you must use the actual name of the global variable. To continue the above example where state is an alias, you cannot:

```
vwait state(foo)
button .b -textvariable state(foo)
```

Instead, you must

```
vwait state$name\(foo)
button .b -textvariable state$name\(foo)
```

The backslash turns off the array reference so Tcl does not try to access name as an array. You do not need to worry about special characters in $name, except parentheses. Once the name has been passed into the Tk widget it will be used directly as a variable name.

CHAPTER 8

Tcl Arrays

This chapter describes Tcl arrays, which provide a flexible mechanism to build many other data structures in Tcl. Tcl command described is: `array`.

An array is a Tcl variable with a string-valued index. You can think of the index as a key, and the array as a collection of related data items identified by different keys. The index, or key, can be any string value. Internally, an array is implemented with a hash table, so the cost of accessing each array element is about the same. Before Tcl 8.0, arrays had a performance advantage over lists that took time to access proportional to the size of the list.

The flexibility of arrays makes them an important tool for the Tcl programmer. A common use of arrays is to manage a collection of variables, much as you use a C struct or Pascal record. This chapter shows how to create several simple data structures using Tcl arrays.

Array Syntax

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 8–1 uses the loop variable value `$i` as an array index. It sets `arr(`*x*`)` to the product of `1 * 2 * ... *` *x*:

Example 8–1 Using arrays.

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

Complex Indices

An array index can be any string, like `orange`, `5`, `3.1415`, or `foo,bar`. The examples in this chapter, and in this book, often use indices that are pretty complex strings to create flexible data structures. As a rule of thumb, you can use any string for an index, but avoid using a string that contains spaces.

Parentheses are not a grouping mechanism.

The main Tcl parser does not know about array syntax. All the rules about grouping and substitution described in Chapter 1 are still the same in spite of the array syntax described here. Parentheses do not group like curly braces or quotes, which is why a space causes problems. If you have complex indices, use a comma to separate different parts of the index. If you use a space in an index instead, then you have a quoting problem. 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.}
set arr(I’m\ asking\ for\ trouble) {I told you so.}
```

If the array index is stored in a variable, then there is no problem with spaces in the variable’s value. The following works well:

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

Array Variables

You can use an array element as you would a simple variable. For example, you can test for its existence with `info exists`, increment its value with `incr`, and append elements to it with `lappend`:

```
if {[info exists stats($event)]} {incr stats($event)}
```

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.

It is an error to use a variable as both an array and a normal variable. The following is an error:

```
set arr(0) 1
set arr 3
=> can’t set "arr": variable is array
```

The name of the array can be the result of a substitution. This is a tricky situation, as shown in Example 8–2:

Example 8–2 Referencing an array indirectly.

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

A better way to deal with this situation is to use the `upvar` command, which is introduced on page 79. The previous example is much cleaner when `upvar` is used:

Example 8–3 Referencing an array indirectly using `upvar`.

```
set name TheArray
=> TheArray
upvar 0 $name a
set a(xyz) {some value}
=> some value
set x $TheArray(xyz)
=> some value
```

The `array` Command

The `array` command returns information about array variables. The `array names` command returns the index names that are defined in the array. If the array variable is not defined, then `array names` just returns an empty list. It allows easy iteration through an array with a `foreach` loop:

```
foreach index [array names arr pattern] {
    # use arr($index)
}
```

The order of the names returned by `array names` is arbitrary. It is essentially determined by the hash table implementation of the array. You can limit what names are returned by specifying a *`pattern`* that matches indices. The pattern is the kind supported by the `string match` command, which is described on page 48.

It is also possible to iterate through the elements of an array one at a time using the search-related commands listed in Table 8–1. The ordering is also random, and 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 a long period of time, then the array search operations might be more appropriate. Frankly, I never use them. Table 8–1 summarizes the `array` command:

Table 8–1 The `array` command.

`array exists` *`arr`*	Returns 1 if *`arr`* is an array variable.
`array get` *`arr ?pattern?`*	Returns a list that alternates between an index and the corresponding array value. *`pattern`* selects matching indices. If not specified, all indices and values are returned.
`array names` *`arr ?pattern`*`?`	Returns the list of all indices defined for *`arr`*, or those that match the string match *`pattern`*.
`array set` *`arr list`*	Initializes the array *`arr`* from *`list`*, which has the same form as the list returned by `array get`.
`array size` *`arr`*	Returns the number of indices defined for *`arr.`*
`array startsearch` *`arr`*	Returns a search token for a search through *`arr`*.
`array nextelement` *`arr id`*	Returns the value of the next element in *`array`* in the search identified by the token *`id`*. Returns an empty string if no more elements remain in the search.
`array anymore` *`arr id`*	Returns 1 if more elements remain in the search.
`array donesearch` *`arr id`*	Ends the search identified by *`id`*.

Converting Between Arrays and Lists

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

```
array set fruit {
   best   kiwi
   worst  peach
   ok     banana
}
array get fruit
=> ok banana best kiwi worst peach
```

Another way to loop through the contents of an array is to use `array get` and the two-variable form of the `foreach` command.

```
foreach {key value} [array get fruit] {
   # key is ok, best, or worst
   # value is some fruit
}
```

Passing Arrays by Name

The `upvar` command works on arrays. You can pass an array name to a procedure and use the `upvar` command to get an indirect reference to the array variable in the caller's scope. This is illustrated in Example 8–4, which inverts an array. As with `array names`, you can specify a pattern to `array get` to limit what part of the array is returned. This example uses `upvar` because the array names are passed into the `ArrayInvert` procedure. The inverse array does not need to exist before you call `ArrayInvert`.

Example 8–4 `ArrayInvert` inverts an array.

```
proc ArrayInvert {arrName inverseName {pattern *}} {
   upvar $arrName array $inverseName inverse
   foreach {index value} [array get array $pattern] {
      set inverse($value) $index
   }
}
```

Building Data Structures with Arrays

This section describes several data structures you can build with Tcl arrays. These examples are presented as procedures that implement access functions to the data structure. Wrapping up your data structures in procedures is good practice. It shields the user of your data structure from the details of its implementation.

Use arrays to collect related variables.

A good use for arrays is to collect together a set of related variables for a module, much as one would use a record in other languages. 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. You can also use `upvar` to manage a collection of arrays, as shown in Example 8–8 on page 95.

Simple Records

Suppose we have a database of information about people. One approach uses a different array for each class of information. The name of the person is the index into each array:

Example 8–5 Using arrays for records, version 1.

```
proc Emp_AddRecord {id name manager phone} {
   global employeeID employeeManager \
      employeePhone employeeName
   set employeeID($name) $id
   set employeeManager($name) $manager
   set employeePhone($name) $phone
   set employeeName($id) $name
}
proc Emp_Manager {name} {
   global employeeManager
   return $employeeManager($name)
}
```

Simple procedures are defined to return fields of the record, which hides the implementation so that you can change it more easily. The employeeName array provides a secondary key. It maps from the employee ID to the name so that the other information can be obtained if you have an ID instead of a name. Another way to implement the same little database is to use a single array with more complex indices:

Example 8–6 Using arrays for records, version 2.

```
proc Emp_AddRecord {id name manager phone} {
   global employee
   set employee(id,$name) $id
   set employee(manager,$name) $manager
   set employee(phone,$name) $phone
   set employee(name,$id) $name
}
proc Emp_Manager {name} {
   global employee
   return $employee(manager,$name)
}
```

The difference between these two approaches is partly a matter of taste. Using a single array can be more convenient because there are fewer variables to manage. In any case, you should hide the implementation in a small set of procedures.

A Stack

A stack can be implemented with either a list or an array. If you use a list, then the push and pop operations have a runtime cost that is proportional to the size of the stack. If the stack has a few elements this is fine. If there are a lot of items in a stack, you may wish to use arrays instead.

Example 8–7 Using a list to implement a stack.

```
proc Push { stack value } {
   upvar $stack list
   lappend list $value
}
proc Pop { stack } {
   upvar $stack list
   set value [lindex $list end]
   set list [lrange $list 0 [expr [llength $list]-2]]
   return $value
}
```

In these examples, the name of the stack is a parameter, and upvar is used to convert that into the data used for the stack. The variable is a list in Example 8–7 and an array in Example 8–8. The user of the stack module does not have to know.

The array implementation of a stack uses one array element to record the number of items in the stack. The other elements of the array have the stack values. The Push and Pop procedures both guard against a nonexistent array with the info exists command. When the first assignment to S(top) is done by Push, the array variable is created in the caller's scope. The example uses array indices in two ways. The top index records the depth of the stack. The other indices are numbers, so the construct $S($S(top)) is used to reference the top of the stack.

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

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

A List of Arrays

Suppose you have many arrays, each of which stores some data, and you want to maintain an overall ordering among the data sets. One approach is to keep a Tcl list with the name of each array in order. Example 8–9 defines RecordInsert to add an array to the list, and an iterator function, RecordIterate, that applies a script to each array in order. The iterator uses upvar to make data an alias for the current array. The script is executed with eval, which is described in detail in Chapter 10. The Tcl commands in script can reference the arrays with the name data:

Example 8–9 A list of arrays.

```
proc RecordAppend {listName arrayName} {
   upvar $listName list
   lappend list $arrayName
}
proc RecordIterate {listName script} {
   upvar $listName list
   foreach arrayName $list {
      upvar #0 $arrayName data
      eval $script
   }
}
```

Another way to implement this list-of-records structure is to keep references to the arrays that come before and after each record. Example 8–10 shows the insert function and the iterator function when using this approach. Once again, `upvar` is used to set up `data` as an alias for the current array in the iterator. In this case, the loop is terminated by testing for the existence of the next array. It is perfectly all right to make an alias with `upvar` to a nonexistent variable. It is also all right to change the target of the `upvar` alias. One detail that is missing from the example is the initialization of the very first record so that its `next` element is the empty string:

Example 8–10 A list of arrays.

```
proc RecordInsert {recName afterThis} {
   upvar $recName record $afterThis after
   set record(next) $after(next)
   set after(next) $recName
}
proc RecordIterate {firstRecord body} {
   upvar #0 $firstRecord data
   while {[info exists data]} {
      eval $body
      upvar #0 $data(next) data
   }
}
```

A Simple In-Memory Database

Suppose you have to manage a lot of records, each of which contain a large chunk of data and one or more key values you use to look up those values. The procedure to add a record is called like this:

`Db_Insert` *`keylist datablob`*

The `datablob` might be a name, value list suitable for passing to `array set`, or simply a large chunk of text or binary data. One implementation of `Db_Insert` might just be:

```
foreach key $keylist {
    lappend Db($key) $datablob
}
```

The problem with this approach is that it duplicates the data chunks under each key. A better approach is to use two arrays. One stores all the data chunks under a simple ID that is generated automatically. The other array stores the association between the keys and the data chunks. Example 8–11, which uses the namespace syntax described in Chapter 14, illustrates this approach. The example also shows how you can easily dump data structures by writing `array set` commands to a file, and then load them later with a `source` command:

Example 8–11 A simple in-memory database.

```
namespace eval db {
   variable data       ;# Array of data blobs
   variable uid 0      ;# Index into data
   variable index      ;# Cross references into data
}
proc db::insert {keylist datablob} {
   variable data
   variable uid
   variable index
   set data([incr uid]) $datablob
   foreach key $keylist {
      lappend index($key) $uid
   }
}
proc db::get {key} {
   variable data
   variable index
   set result {}
   if {![info exist index($key)]} {
      return {}
   }
   foreach uid $index($key) {
      lappend result $data($uid)
   }
   return $result
}
proc db::save {filename} {
   variable uid
   set out [open $filename w]
   puts $out [list namespace eval db \
      [list variable uid $uid]]
   puts $out [list array set db::data [array get db::data]]
   puts $out [list array set db::index [array get db::index]]
   close $out
}
proc db::load {filename} {
   source $filename
}
```

CHAPTER 9

Working with Files and Programs

This chapter describes how to run programs, examine the file system, and access environment variables through the env array. Tcl commands described are: exec, file, open, close, read, write, puts, gets, flush, seek, tell, glob, pwd, cd, exit, pid, and registry.

This chapter describes how to run programs and access the file system from Tcl. These commands were designed for UNIX. In Tcl 7.5 they were implemented in the Tcl ports to Windows and Macintosh. There are facilities for naming files and manipulating file names in a platform-independent way, so you can write scripts that are portable across systems. 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 Programs with exec

The exec command runs 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 channel or exits with a nonzero status code, then exec raises an error. If you do not care about the exit status, or you use a program that insists on writing to standard error, then you can use catch to mask the errors:

```
catch {exec program arg arg} result
```

* Unlike other UNIX shell exec commands, 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.

The `exec` command supports a full set of *I/O redirection* and *pipeline* syntax. Each process normally has three I/O channels associated with it: standard input, standard output, and standard error. With I/O redirection, you can divert these I/O channels to files or to I/O channels you have opened with the Tcl `open` command. A pipeline is a chain of 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 9–1 Using `exec` on a process pipeline.

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

Example 9–1 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 the lines. The error output of the command is diverted to the null device to suppress any error messages. Table 9–1 provides a summary of the syntax understood by the `exec` command.

Table 9–1 Summary of the `exec` syntax for I/O redirection.

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

A trailing & causes the program to run in the background. In this case, the process identifier 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.

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 channels by using @.

The auto_noexec Variable

The Tcl shell programs are set up during interactive use to attempt to execute unknown Tcl commands as 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
```

Limitations of exec on Windows

Windows 3.1 has an unfortunate combination of special cases that stem from console-mode programs, 16-bit programs, and 32-bit programs. In addition, pipes are really just simulated by writing output from one process to a temporary file and then having the next process read from that file. If exec or a process pipeline fails, it is because of a fundamental limitation of Windows. The good news is that Windows 95 and Windows NT cleaned up most of the problems with exec. Windows NT 4.0 is the most robust.

Tcl 8.0p2 was the last release to officially support Windows 3.1. That release includes Tcl1680.dll, which is necessary to work with the win32s subsystem. If you copy that file into the same directory as the other Tcl DLLs, you may be able to use later releases of Tcl on Windows 3.1. However, there is no guarantee this trick will continue to work.

AppleScript on Macintosh

The exec command is not provided on the Macintosh. Tcl ships with an AppleScript extension that lets you control other Macintosh applications. You can find documentation in the AppleScript.html that goes with the distribution. You must use package require to load the AppleScript command:

```
package require Tclapplescript
AppleScript junk
=> bad option "junk": must be compile, decompile, delete,
execute, info, load, run, or store.
```

The file Command

The file command provides several ways to check the status of files in the file system. For example, you can find out if a file exists, what type of file it is, and other file attributes. There are facilities for manipulating files in a platform-independent manner. Table 9–2 provides a summary of the various forms of the file command. They are described in more detail later. Note that the split, join, and pathtype operations were added in Tcl 7.5. The copy, delete, mkdir, and rename operations were added in Tcl 7.6. The attributes operation was added in Tcl 8.0.

Table 9–2 The file command options.

file atime *name*	Returns access time as a decimal string.
file attributes *name* ?*option*? ?*value*? ...	Queries or sets file attributes. (Tcl 8.0)
file copy ?-force? *source destination*	Copies file *source* to file *destination*. The *source* and *destination* can be directories. (Tcl 7.6)
file delete ?-force? *name*	Deletes the named file. (Tcl 7.6)
file dirname *name*	Returns parent directory of file *name*.
file executable *name*	Returns 1 if *name* has execute permission, else 0.
file exists *name*	Returns 1 if *name* exists, else 0.
file extension *name*	Returns the part of *name* from the last dot (i.e., .) to the end. The dot is included in the return value.
file isdirectory *name*	Returns 1 if *name* is a directory, else 0.
file isfile *name*	Returns 1 if *name* is not a directory, symbolic link, or device, else 0.
file join *path path*...	Joins pathname components into a new pathname. (Tcl 7.5)
file lstat *name var*	Places attributes of the link *name* into *var*.
file mkdir *name*	Creates directory *name*. (Tcl 7.6)
file mtime *name*	Returns modify time of *name* as a decimal string.

Table 9–2 The `file` command options. (Continued)

`file nativename` *`name`*	Returns the platform-native version of *`name`*. (Tk 8.0).
`file owned` *`name`*	Returns 1 if current user owns the file *`name`*, else 0.
`file pathtype` *`name`*	`relative`, `absolute`, or `driverelative`. (Tcl 7.5)
`file readable` *`name`*	Returns 1 if *`name`* has read permission, else 0.
`file readlink` *`name`*	Returns the contents of the symbolic link *`name`*.
`file rename ?-force?` *`old new`*	Changes the name of *`old`* to *`new`*. (Tcl 7.6)
`file rootname` *`name`*	Returns all but the extension of *`name`* (i.e., up to but not including the last `.` in *`name`*).
`file size` *`name`*	Returns the number of bytes in *`name`*.
`file split` *`name`*	Splits *`name`* into its pathname components. (Tcl 7.5)
`file stat` *`name var`*	Places attributes of *`name`* into array *`var`*. The elements defined for *`var`* are listed in Table 9–3.
`file tail` *`name`*	Returns the last pathname component of *`name`*.
`file type` *`name`*	Returns type identifier, which is one of: `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.
`file writable` *`name`*	Returns 1 if *`name`* has write permission, else 0.

Cross-Platform File Naming

Files are named differently on UNIX, Windows, and Macintosh. UNIX separates file name components with a forward slash (/), Macintosh separates components with a colon (:), and Windows separates components with a backslash (\). In addition, the way that absolute and relative names are distinguished is different. For example, these are absolute pathnames for the Tcl script library (i.e., `$tcl_library`) on Macintosh, Windows, and UNIX, respectively:

```
Disk:System Folder:Extensions:Tool Command Language:tcl7.6
c:\Program Files\Tcl\lib\Tcl7.6
/usr/local/tcl/lib/tcl7.6
```

The good news is that Tcl provides operations that let you deal with file pathnames in a platform-independent manner. The file operations described in this chapter allow either native format or the UNIX naming convention. The backslash used in Windows pathnames is especially awkward because the backslash is special to Tcl. Happily, you can use forward slashes instead:

```
c:/Program Files/Tcl/lib/Tcl7.6
```

There are some ambiguous cases that can be specified only with native pathnames. On my Macintosh, Tcl and Tk are installed in a directory that has a slash in it. You can name it only with the native Macintosh name:

```
Disk:Applications:Tcl/Tk 4.2
```

Another construct to watch out for is a leading `//` in a file name. This is the Windows syntax for network names that reference files on other computers. You can avoid accidentally constructing a network name by using the `file join` command described next. Of course, you can use network names to access remote files.

If you must communicate with external programs, you may need to construct a file name in the native syntax for the current platform. You can construct these names with `file join` described later. You can also convert a UNIX-like name to a native name with `file nativename`.

Several of the `file` operations operate on pathnames as opposed to returning information about the file itself. You can use the `dirname`, `extension`, `join`, `pathtype`, `rootname`, `split`, and `tail` operations on any string; there is no requirement that the pathnames refer to an existing file.

Building up Pathnames: `file join`

You can get into trouble if you try to construct file names by simply joining components with a slash. If part of the name is in native format, joining things with slashes will result in incorrect pathnames on Macintosh and Windows. The same problem arises when you accept user input. The user is likely to provide file names in native format. For example, this construct will not create a valid pathname on the Macintosh because `$tcl_library` is in native format:

```
set file $tcl_library/init.tcl
```

Use `file join` to construct file names.

The platform-independent way to construct file names is with `file join`. The following command returns the name of the `init.tcl` file in native format:

```
set file [file join $tcl_library init.tcl]
```

The `file join` operation can join any number of pathname components. In addition, it has the feature that an absolute pathname overrides any previous components. For example (on UNIX), `/b/c` is an absolute pathname, so it overrides any paths that come before it in the arguments to `file join`:

```
file join a b/c d
=> a/b/c/d
file join a /b/c d
=> /b/c/d
```

On Macintosh, a relative pathname starts with a colon, and an absolute pathname does not. To specify an absolute path, you put a trailing colon on the first component so that it is interpreted as a volume specifier. These relative components are joined into a relative pathname:

```
file join a :b:c d
=> :a:b:c:d
```

In the next case, `b:c` is an absolute pathname with `b:` as the volume specifier. The absolute name overrides the previous relative name:

```
file join a b:c d
=> b:c:d
```

The file join operation converts UNIX-style pathnames to native format. For example, on Macintosh you get this:

```
file join /usr/local/lib
=> usr:local:lib
```

Chopping Pathnames: split, dirname, tail

The file split command divides a pathname into components. It is the inverse of file join. The split operation detects automatically if the input is in native or UNIX format. The results of file split may contain some syntax to help resolve ambiguous cases when the results are passed back to file join. For example, on Macintosh a UNIX-style pathname is split on slash separators. The Macintosh syntax for a volume specifier (Disk:) is returned on the leading component:

```
file split "/Disk/System Folder/Extensions"
=> Disk: {System Folder} Extensions
```

A common reason to split up pathnames is to divide a pathname into the directory part and the file part. This task is handled directly by the dirname and tail operations. The dirname operation returns the parent directory of a pathname, while tail returns the trailing component of the pathname:

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

For a pathname with a single component, the dirname option returns ".", on UNIX and Windows, or ":" on Macintosh. This is the name of the current directory.

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 including the period.) The root option returns everything up to, but not including, the last period in the pathname:

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

Manipulating Files and Directories

Tcl 7.6 added file operations to copy files, delete files, rename files, and create directories. In earlier versions it was necessary to exec other programs to do these things, except on Macintosh, where cp, rm, mv, mkdir, and rmdir were built in. These commands are no longer supported on the Macintosh. Your scripts should use the file command operations described below to manipulate files in a platform-independent way.

File name patterns are not directly supported by the file operations. Instead, you can use the glob command described on page 115 to get a list of file names that match a pattern.

Copying Files

The file copy operation copies files and directories. The following example copies *file1* to *file2*. If *file2* already exists, the operation raises an error unless the -force option is specified:

```
file copy ?-force? file1 file2
```

Several files can be copied into a destination directory. The names of the source files are preserved. The -force option indicates that files under *directory* can be replaced:

```
file copy ?-force? file1 file2 ... directory
```

Directories can be recursively copied. The -force option indicates that files under *dir2* can be replaced:

```
file copy ?-force? dir1 dir2
```

Creating Directories

The file mkdir operation creates one or more directories:

```
file mkdir dir dir ...
```

It is *not* an error if the directory already exists. Furthermore, intermediate directories are created if needed. This means that you can always make sure a directory exists with a single mkdir operation. Suppose /tmp has no subdirectories at all. The following command creates /tmp/sub1 and /tmp/sub1/sub2:

```
file mkdir /tmp/sub1/sub2
```

The -force option is not understood by file mkdir, so the following command accidentally creates a folder named -force, as well as one named oops.

```
file mkdir -force oops
```

Deleting Files

The file delete operation deletes files and directories. It is *not* an error if the files do not exist. A non-empty directory is not deleted unless the -force option is specified, in which case it is recursively deleted:

```
file delete ?-force? name name ...
```

To delete a file or directory named -force, you must specify a nonexistent file before the -force to prevent it from being interpreted as a flag (-force -force won’t work):

```
file delete xyzzy -force
```

Renaming Files and Directories

The `file rename` operation changes a file's name from *`old`* to *new*. The `-force` option causes *new* to be replaced if it already exists.

```
file rename ?-force? old new
```

Using `file rename` is the best way to update an existing file. First, generate the new version of the file in a temporary file. Then, use `file rename` to replace the old version with the new version. This ensures that any other programs that access the file will not see the new version until it is complete.

File Attributes

There are several file operations that return specific file attributes: `atime`, `executable`, `exists`, `isdirectory`, `isfile`, `mtime`, `owned`, `readable`, `readlink`, `size` and `type`. Refer to Table 9–2 on page 102 for their function. The following command uses `file mtime` to compare the modify times of two files. 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 will appreciate this example:

Example 9–2 Comparing file modify times.

```
proc newer { file1 file2 } {
   if ![file exists $file2] {
      return 1
   } else {
      # Assume file1 exists
      expr [file mtime $file1] > [file mtime $file2]
   }
}
```

The `stat` and `lstat` operations return a collection of file attributes. They take a third argument that is the name of an array variable, and they initialize that array with elements that contain the file attributes. If the file is a symbolic link, then the `lstat` operation returns information about the link itself and the `stat` operation returns information about the target of the link. The array elements are listed in Table 9–3. All the element values are decimal strings, except for `type`, which can have the values returned by the `type` option. The element names are based on the UNIX `stat` system call. Use the `file attributes` command described later to get other platform-specific attributes:

Table 9–3 Array elements defined by `file stat`.

`atime`	The last access time, in seconds.
`ctime`	The last change time (not the create time), in seconds.
`dev`	The device identifier, an integer.
`gid`	The group owner, an integer.
`ino`	The file number (i.e., inode number), an integer.
`mode`	The permission bits.
`mtime`	The last modify time, in seconds.
`nlink`	The number of links, or directory references, to the file.
`size`	The number of bytes in the file.
`type`	`file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.
`uid`	The owner's user ID, an integer.

Example 9–3 uses the device (`dev`) and inode (`ino`) attributes of a file to determine whether two pathnames reference the same file. The attributes are UNIX specific; they are not well defined on Windows and Macintosh.

Example 9–3 Determining whether 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 `file attributes` operation was added in Tcl 8.0 to provide access to platform-specific attributes. The `attributes` operation lets you set and query attributes. The interface uses option-value pairs. With no options, all the current values are returned.

```
file attributes book.doc
=> -creator FRAM -hidden 0 -readonly 0 -type MAKR
```

These Macintosh attributes are explained in Table 9–4. The four-character type codes used on Macintosh are illustrated on page 514. With a single option, only that value is returned:

```
file attributes book.doc -readonly
=> 0
```

The attributes are modified by specifying one or more option–value pairs. Setting attributes can raise an error if you do not have the right permissions:

```
file attributes book.doc -readonly 1 -hidden 0
```

Table 9–4 Platform-specific file attributes.

`-permissions` *`mode`*	File permission bits. *`mode`* is a number with bits defined by the `chmod` system call. (UNIX)
`-group` *`ID`*	The group owner of the file. (UNIX)
`-owner` *`ID`*	The owner of the file. (UNIX)
`-archive` *`bool`*	The archive bit, which is set by backup programs. (Windows)
`-hidden` *`bool`*	If set, then the file does not appear in listings. (Windows, Macintosh)
`-readonly` *`bool`*	If set, then you cannot write the file. (Windows, Macintosh)
`-system` *`bool`*	If set, then you cannot remove the file. (Windows)
`-creator` *`type`*	*`type`* is 4-character code of creating application. (Macintosh)
`-type` *`type`*	*`type`* is 4-character type code. (Macintosh)

Input/Output Command Summary

The following sections describe how to open, read, and write files. The basic model is that you open a file, read or write it, then close the file. Network sockets also use the commands described here. Socket programming is discussed in Chapter 17, and more advanced *event-driven* I/O is described in Chapter 16. Table 9–5 lists the basic commands associated with file I/O:

Table 9–5 Tcl commands used for file access.

`open` *`what ?access? ?permissions?`*	Returns channel ID for a file or pipeline.
`puts ?-nonewline?` *`?channel? string`*	Writes a string.
`gets` *`channel ?varname?`*	Reads a line.
`read` *`channel ?numBytes?`*	Reads *`numBytes`* bytes, or all data.
`read -nonewline` *`channel`*	Reads all bytes and discard the last `\n`.
`tell` *`channel`*	Returns the seek offset.
`seek` *`channel offset ?origin?`*	Sets the seek offset. *`origin`* is one of `start`, `current`, or `end`.
`eof` *`channel`*	Queries end-of-file status.
`flush` *`channel`*	Writes buffers of a channel.
`close` *`channel`*	Closes an I/O channel.

Opening Files for I/O

The `open` command sets up an I/O channel to either a file or a pipeline of processes. The return value of `open` is an identifier for the I/O channel. Store the result of `open` in a variable and use the variable as you used the `stdout`, `stdin`, and `stderr` identifiers in the examples so far. 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 9–6 summarizes the first form, while Table 9–7 summarizes the POSIX flags. If *`access`* is not specified, it defaults to read.

Example 9–4 Opening a file for writing.

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

Table 9–6 Summary of the `open` access arguments.

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

Table 9–7 Summary of `POSIX` flags for the access argument.

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

The *`permissions`* argument is a value used for the permission bits on a newly created file. UNIX uses three bits each for the owner, group, and everyone else. The bits specify read, write, and execute permission. These bits are usually specified with an octal number, which has a leading zero, so that there is one octal digit for each set of bits. The default permission bits are `0666`, which grant read/write access to everybody. Example 9–4 specifies `0600` so that the file is readable and writable only by the owner. `0775` would grant read, write, and execute permissions to the owner and group, and read and execute permissions to everyone else. You can set other special properties with additional high-order bits. Consult the UNIX manual page on *chmod* command for more details.

The following example illustrates how to use a list of POSIX access flags to open a file for reading and writing, creating it if needed, and not truncating it. This is something you cannot do with the simpler form of the access argument:

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

Catch errors from open.

In general, you should 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 9–5 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

You can open a process pipeline 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 `open` returns. The following example runs the UNIX *sort* program on the password file, and it uses the `split` command to separate the output lines into list elements:

Example 9–6 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. 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 the data out to the spawned processes before you try to read any output from the pipeline. You can also use the `fconfigure` command described on page 223 to force line buffering. Remember that read-write pipes will not work at all with Windows 3.1 because pipes are simulated with files. Event-driven I/O is also very useful with pipes. It means you can do other processing while the pipeline executes, and simply respond when the pipe generates data. This is described in Chapter 16.

Expect

If you are trying to do sophisticated things with an external application, you will find that the *Expect* extension provides a much more powerful interface than a process pipeline. *Expect* adds Tcl commands that are used to control interactive applications. It is extremely useful for automating FTP, Telnet, and programs under test. It comes as a Tcl shell named *expect*, and it is also an extension that you can dynamically load into other Tcl shells. It was created by Don Libes at the National Institute of Standards and Technology (NIST). *Expect* is described in *Exploring Expect* (Libes, O'Reilly & Associates, Inc., 1995). You can find the software on the CD and on the web at:

```
http://expect.nist.gov/
```

Reading and Writing

The standard I/O channels are already open for you. There is a standard input channel, a standard output channel, and a standard error output channel. These channels are identified by `stdin`, `stdout`, and `stderr`, respectively. Other I/O channels are returned by the `open` command, and by the `socket` command described on page 226.

There may be cases when the standard I/O channels are not available. Windows has no standard error channel. Some UNIX window managers close the standard I/O channels when you start programs from window manager menus. You can also close the standard I/O channels with `close`.

The `puts` and `gets` Commands

The `puts` command writes a string and a newline to the output channel. There are a couple of details about the `puts` command that we have not yet used. It takes a `-nonewline` argument that prevents the newline character that is normally appended to the output channel. This is used in the prompt example below. The second feature is that the channel identifier is optional, defaulting to `stdout` if not specified. Note that you must use `flush` to force output of a partial line. This is illustrated in Example 9–7.

Example 9–7 Prompting for input.

```
puts -nonewline "Enter value: "
flush stdout ;# Necessary to get partial line output
set answer [gets stdin]
```

The `gets` command reads a line of input, and it has two forms. In the previous example, with just a single argument, `gets` returns the line read from the specified I/O channel. It discards the trailing newline from the return value. If end of file is reached, an empty string is returned. You must 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 a named variable and returns the number of bytes read. It discards the trailing newline, which is not counted. A -1 is returned if the channel has reached the end of file.

Example 9–8 A read loop using `gets`.

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

The `read` Command

The `read` command reads blocks of data, and this capability is often more efficient. There are two forms for `read`: You can specify the `-nonewline` argument or the *`numBytes`* argument, but not both. Without *`numBytes`*, the whole file (or what is left in the I/O channel) 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 there is not enough data in the channel. The trailing newline is not discarded in this case.

Example 9–9 A read loop using `read` and `split`.

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

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

Platform-Specific End of Line Characters

Tcl automatically detects different end of line conventions. On UNIX, text lines are ended with a newline character (\n). On Macintosh, they are terminated with a carriage return (\r). On Windows, they are terminated with a carriage return, newline sequence (\r\n). Tcl accepts any of these, and the line terminator can even change within a file. All these different conventions are converted to the UNIX style so that once read, text lines are always terminated with a newline character (\n). Both the read and gets commands do this conversion.

During output, text lines are generated in the platform-native format. The automatic handling of line formats means that it is easy to convert a file to native format. You just need to read it in and write it out:

```
puts -nonewline $out [read $in]
```

To suppress conversions, use the fconfigure command, which is described in more detail on page 223.

Example 9–10 demonstrates a File_Copy procedure that translates files to native format. It is complicated because it handles directories:

Example 9–10 Copy a file and translate to native format.

```
proc File_Copy {src dest} {
   if [file isdirectory $src] {
      file mkdir $dest
      foreach f [glob -nocomplain [file join $src *]] {
         File_Copy $f [file join $dest [file tail $f]]
      }
      return
   }
   if [file isdirectory $dest] {
      set dest [file join $dest [file tail $src]]
    }
   set in [open $src]
   set out [open $dest w]
   puts -nonewline $out [read $in]
   close $out ; close $in
}
```

Random Access I/O

The seek and tell commands provide random access to I/O channels. Each channel 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 sets the seek offset by an amount, which can be positive or negative, from an origin which is either start, current, or end.

Closing I/O Channels

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

The `close` command can raise an error.

If the channel was a process pipeline and any of the processes wrote to their standard error channel, then Tcl believes this is an error. The error is raised when the channel to the pipeline is finally closed. Similarly, if any of the processes in the pipeline exit with a nonzero status, `close` raises an error.

The Current Directory — cd and pwd

Every process has a current directory that is used as the starting point when resolving a relative pathname. The `pwd` command returns the current directory, and the `cd` command changes the current directory. Example 9–11 uses these commands.

Matching File Names with `glob`

The `glob` command expands a pattern into the set of matching file names. The general form of the `glob` command is:

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

The pattern syntax is similar to the `string match` patterns:

- `*` matches zero or more characters.
- `?` matches a single character.
- `[abc]` matches a set of characters.
- `{a,b,c}` matches any of `a`, `b`, or `c`.
- All other characters must match themselves.

The `-nocomplain` flag causes `glob` to return an empty list if no files match the pattern. Otherwise, `glob` raises an error if no files match.

The `--` flag must be used if the `pattern` begins with a `-`.

Unlike the glob matching in *csh*, the Tcl `glob` command matches only the names of existing files. In *csh*, the `{a,b}` construct can match nonexistent names. In addition, the results of `glob` are not sorted. Use the `lsort` command to sort its result if you find it important.

Example 9–11 shows the `FindFile` procedure, which 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` guards against bogus names. The `glob` command matches file names:

Example 9–11 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 [file join $startDir $match]
   }
   foreach file [glob -nocomplain *] {
      if [file isdirectory $file] {
         FindFile [file join $startDir $file] $namePat
      }
   }
   cd $pwd
}
```

Expanding Tilde in File Names

The `glob` command also expands a leading tilde (~) in filenames. There are two cases:

- `~/` expands to the current user's home directory.
- `~`*`user`* expands to the home directory of *`user`*.

If you have a file that starts with a literal tilde, you can avoid the tilde expansion by adding a leading `./` (e.g., `./~foobar`).

The `exit` and `pid` Commands

The `exit` command terminates your script. Note that `exit` causes termination of the whole process that was running the script. 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 changes each time you run your script. It is also common to embed the process ID in the name of temporary files.

You can also find out the process IDs associated with a process pipeline with `pid`:

```
set pipe [open "|command"]
set pids [pid $pipe]
```

There is no built-in mechanism to control processes in Tcl. On UNIX systems you can exec the *kill* program to terminate a process:

```
exec kill $pid
```

Environment Variables

Environment variables are a collection of string-valued variables associated with each process. The process's 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, they result in changes to the corresponding environment variable. Environment variables are inherited by child processes, so programs run with the exec command inherit the environment of the Tcl script. The following example prints the values of environment variables.

Example 9–12 Printing environment variable values.

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

Note: Environment variables can be initialized for Macintosh applications by editing a resource of type STR# whose name is Tcl Environment Variables. This resource is part of the *tclsh* and *wish* applications. Follow the directions on page 28 for using *ResEdit*. The format of the resource values is *NAME=VALUE*.

The registry Command

Windows uses the *registry* to store various system configuration information. The Windows tool to browse and edit the registry is called *regedit*. Tcl provides a registry command. It is a loadable package that you must load by using:

```
package require registry
```

The registry structure has keys, value names, and typed data. The value names are stored under a key, and each value name has data associated with it.

The keys are organized into a hierarchical naming system, so another way to think of the value names is as an extra level in the hierarchy. The main point is that you need to specify both a key name and a value name in order to get something out of the registry. The key names have one of the following formats:

```
\\hostname\rootname\keypath
rootname\keypath
rootname
```

The *rootname* is one of HKEY_LOCAL_MACHINE, HKEY_PERFORMANCE_DATA, HKEY_USERS, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_CURRENT_CONFIG, or HKEY_DYN_DATA. Tables 9–8 and 9–9 summarize the registry command and data types:

Table 9–8 The registry command.

registry delete *key* ?*valueName*?	Deletes the *key* and the named value, or it deletes all values under the key if *valueName* is not specified.
registry get *key* *valueName*	Returns the value associated with *valueName* under *key*.
registry keys *key* ?*pat*?	Returns the list of keys or value names under *key* that match *pat*, which is a string match pattern.
registry set *key*	Creates *key*.
registry set *key* *valueName data* ?*type*?	Creates *valueName* under *key* with value *data* of the given *type*. Types are listed in Table 9–9.
registry type *key* *valueName*	Returns the type of *valueName* under *key*.
registry values *key* ?*pat*?	Returns the names of the values stored under *key* that match *pat*, which is a string match pattern.

Table 9–9 The registry data types.

binary	Arbitrary binary data.
none	Arbitrary binary data.
expand_sz	A string that contains references to environment variables with the %VARNAME% syntax.
dword	A 32-bit integer.
dword_big_endian	A 32-bit integer in the other byte order. It is represented in Tcl as a decimal string.
link	A symbolic link.
multi_sz	An array of strings, which are represented as a Tcl list.
resource_list	A device driver resource list.