CHAPTER

3 Introduction to the Tcl Language

The next five chapters constitute a Tcl language tutorial. This chapter provides an overview of the Tcl syntax, data structures, and enough commands to develop applications. Chapter 4 discusses Tcl I/O support for files, pipes, and sockets. Chapters 5–8 introduce more commands and techniques and provide examples showing how Tcl data constructs can be used to create complex data constructs such as structures and trees. Chapters 9 and 10 introduce the TclOO object-oriented support package and explain some tricks in using dynamic and introspective object-oriented programming effectively.

This introduction to the Tcl language will give you an overview of how to use Tcl, rather than be a complete listing of all commands and all options. The on-line reference pages are the complete reference for the commands. See Chapter 1 for a discussion on how to access the on-line help on UNIX, Macintosh, and Windows platforms. The companion website contains a Tcl/Tk reference guide that contains brief listings of all commands and all options.

If you prefer a more extensive tutorial, see the *tutorials* list on the companion website. You will find a copy of `TclTutor`, a computer-assisted instruction program that covers all of the commands in Tcl, and most of the command options.

Chapters 11 through 14 constitute the Tk tutorial. If you are performing graphics programming, you may be tempted to skip ahead to those chapters and just read about the GUIs. Don't do it! Tcl is the glue that holds the graphic widgets together. Tk and the other Tcl extensions build on the Tcl foundation. If you glance ahead for the Tk tutorial, plan on coming back to fill in the gaps.

This book will print the command syntax using the font conventions used by the Tcl on-line manual and help pages. This convention is as follows.

`commandname`	The command name appears first in this type font.
`subcommandname`	If the command supports subcommands, they will also be in this type font.
`-option`	Options appear in italics. The first character is a dash (`-`).
`argument`	Arguments to a command appear in italics.
`?-option?`	Options that are not required are bounded by question marks.
`?argument?`	Arguments that are not required are bounded by question marks.

The following is an example.

Syntax: `puts` *`?-nonewline? ?channel? outputString`*

The command name is `puts`. The `puts` command will accept the options *`-nonewline`* and *`channel`* as arguments, and must include an `outputString` argument.

3.1 OVERVIEW OF THE BASICS

The Tcl language has a simple and regular syntax. You can best approach Tcl by learning the overall syntax and then learning the individual commands. Because all Tcl extensions use the same base interpreter, they all use the same syntax. This consistency makes it easy to learn new sets of commands when the need arises.

3.1.1 Syntax

Tcl is a position-based language, not a keyword-based language. Unlike languages such as C, FORTRAN, and Java, there are no reserved strings. The first word of a Tcl command line must always be a Tcl command; either a built-in command, a procedure name, or (when `tclsh` is in interactive mode) an external command.

A complete Tcl command is a list of words. The first word must be a Tcl command name or a procedure name. The words that follow may be subcommands, options, or arguments to the command. The command is terminated by a newline or a semicolon.

For example, the word `puts` at the beginning of a command is a command name, but `puts` in the second position of a command could be a subcommand or a variable name. The Tcl interpreter keeps separate hash tables for the command names and the variable names, so you can have both a command `puts` and a variable named `puts` in the same procedure. The Tcl syntax rules are as follows.

- The first word of a command line is a command name.
- Each word in a command line is separated from the other words by one or more spaces.
- Words can be grouped with double quotes or curly braces.
- A list can be ungrouped with the three-character {*} operator.
- Commands are terminated with a newline or semicolon.
- A word starting with a dollar sign ($) must be a variable name. This string will be replaced by the value of the variable.
- A variable name followed by a value within parentheses (no spaces) is an associative array: `arrayName(index)`
- Words enclosed within square brackets must be a legal Tcl command. This string will be replaced by the results of evaluating the command.

The Tcl interpreter treats a few characters as having special meaning. These characters are as follows.

Substitution Symbols	
$	The word following the dollar sign must be a variable name. Tcl will substitute the value assigned to that variable for the `$varName` string.
[]	The words between the square brackets must be a Tcl command string. The Tcl interpreter will evaluate the string as a command. The value returned by the command will replace the brackets and string.

Grouping Symbols	
””	Groups multiple words into a single string. Substitutions will occur within this string.
{}	Groups multiple words into a single string. No special character interpretation will occur within this string. A newline within curly braces does not denote the end of a command, and no variable or command substitutions will occur.
Other	
\	Escapes the single character following the backslash. This character will not be treated as a special character. This can be used to escape a dollar sign to inhibit substitution, or to escape a newline character to continue a command across multiple lines.
;	Marks the end of a command.
`<newline>`	Marks the end of a command.
#	Marks the rest of the line as a comment. Note that the # must be in a position where a Tcl command name could be: either the first character on a line or following a semicolon (;).

Example 1

```
x=4
```

Not valid: The string `x=4` is interpreted as the first word on a line and will be evaluated as a procedure or command name. This is not an assignment statement.

`Error Message:   invalid command name "x=4"`

```
puts "This command has one argument";
```

Valid: This is a complete command.

```
puts one; puts two;
```

Valid: This line has two commands.

```
puts one puts two
```

Not valid: The first `puts` command is not terminated with a semicolon, so Tcl interprets the line as a `puts` command with three arguments.

`Error Message:   bad argument "two": should be "nonewline"`

3.1.2 Grouping Words

The spaces between words are important. Since Tcl does not use keywords, it scans commands by checking for symbols separated by white space. Tcl uses spaces to determine which words are commands, subcommands, options, or data. If a data string has multiple words that must be treated as a single set of data, the string must be grouped with quotes (“ ”) or curly braces ({}).

Example 2

```
if { $x > 2} {
  set greater true
}
```

Valid: If the value of `x` is greater than 2, the value of `greater` is set to ”true.”

```
if{ $x > 2} {
  set greater true
}
```

Not valid: No space between `if` and test left brace.
`Error Message: invalid command name "if{"`

```
if {$x > 2}{
  set greater true
}
```

Not valid: No space between test and body left brace.
`Error Message: invalid command name "}"`

```
set x "a b"
```

Valid: The variable `x` is assigned the value `a b`.

```
set x {a b}
```

Valid: The variable `x` is assigned the value `a b`.

```
set x a b
```

Not valid: Too many arguments to set.
`Error Message: wrong # args: should be "set varName ?newValue?"`

The Tcl interpreter treats quotes and braces differently. These differences are discussed later in this chapter, and in more detail in Chapter 4.

3.1.3 Comments

A comment is denoted by putting a pound sign (#) in the position where a command name could be. The *Tcl Style Guide* recommends that this be the first character on a line, but the pound sign could be the first character after a semicolon.

Example 3

```
# This is a comment
```

Valid: This is a valid comment.

```
puts "test" ; # Comment after a command.
```

Valid: But not Recommended style.

```
puts "test" # this is a syntax error.
```

Not valid: The `puts` command was not terminated.

3.1.4 Data Representation

Tcl does not require that you declare variables before using them. The first time you assign a value to a variable name, the Tcl interpreter allocates space for the data and adds the variable name to the internal tables.

A variable name is an arbitrarily long sequence of letters, numbers, or punctuation characters. Although any characters (including spaces) can be used in variable names, the convention is to follow naming rules similar to those in C and Pascal; start a variable name with a letter, followed by a sequence of alphanumeric characters.

The usual convention is to start variable names with a lowercase letter.

The *Tcl Style Guide* recommends that you start variable and procedure names that are exported from a namespace with a lowercase letter and start variable and procedure names that are only for internal use with an uppercase letter. The rationale for this is that items that are intended for internal use should require more keystrokes than items intended for external use. This document is available at the Tcl/Tk resource (*www.tcl.tk/doc/styleGuide.pdf*) and on the companion website.

A variable is referenced by its name. Placing a dollar sign (`$`) in front of the variable name causes the Tcl interpreter to replace the `$`*`varName`* string with the value of that variable. The Tcl interpreter always represents the value of a Tcl variable as a printable string within your script. (Internally, it may be a floating-point value or an integer. Tcl interpreter internals are described in Chapter 15.)

Example 4

`set x four`	Set the value of a variable named `x` to four.
`set pi 3.14`	Set the value of a variable named `pi` to 3.14.
`puts "pi is $pi"`	Display the string: ”pi is 3.14”.
`set pi*2 6.28`	Set the value of a variable named `pi*2` to 6.28.
`set "bad varname" "Don’t Do This"`	Set the value of a variable named `bad varname` to `Don’t Do This`.

Note that the `*` symbol in the variable name `pi*2` does not mean to multiply. Since the `*` is embedded in a word, it is simply another character in a variable name. The last example shows how spaces can be embedded in a variable name. This is not recommended style.

3.1.5 Command Results

All Tcl commands return either a data value or an empty string. The data can be assigned to a variable, used in a conditional statement such as an `if`, or passed to another command.

The Tcl interpreter will evaluate a command enclosed within square brackets immediately, and replace that string with the value returned when the command is evaluated. This is the same as putting a command inside backquotes in UNIX shell programming.

For example, the `set` command always returns the current value of the variable being assigned a value. In the example that follows, when `x` is assigned the value of `"apple"`, the `set` command returns `"apple"`. When the command `set x "pear"` is evaluated within the square brackets, it returns `"pear"`, which is then assigned to the variable `y`.

Example 5

```
# The set x command returns the contents of the variable.
%  set x "apple"
apple
%  set y [set x "pear"]
pear
%  puts $y
pear
%  puts $x
pear
```

In the previous example, the quotes around the words `apple` and `pear` are not required by the Tcl interpreter. However, it is good practice to place strings within quotes.

3.1.6 Errors

Like other modern languages, Tcl has separate mechanisms for the status and data returns from commands and functions. If a Tcl command fails to execute for a syntactic reason (incorrect arguments, and so on), the interpreter will generate an error and invoke an error handler. The default error handler will display a message about the cause of the error and stop evaluating the current Tcl script.

A script can disable the default error handling by catching the error with the `catch` command, and can generate an error with the `error` command.

The `catch` command is used to catch an error condition without causing a script to abort processing.

Syntax: `catch` *`script ?varName?`*

`script` A script to evaluate

`?varName?` An optional variable name to receive the results of evaluating the script.

If the script generates an error, the `catch` command will return a true (1). If the script runs without error, the `catch` command will return a false (0). If an optional variable name is provided, the return value from running the command (either an error message or a return value) is set as that variable's value.

3.2 COMMAND EVALUATION AND SUBSTITUTIONS

Much of the power of the Tcl language is in the mechanism used to evaluate commands. The evaluation process is straightforward and elegant but, like a game of Go, it can catch you by surprise if you do not understand how it works.

Tcl processes commands in two steps. First, it performs command and variable substitutions, and then it evaluates the resulting string. Note that everything goes through this evaluation procedure. Both internal commands (such as `set`) and subroutines you write are processed by the same evaluation code. A `while` command, for example, is treated just like any other command. It takes two arguments: a test and a body of code to execute if the test is true.

3.2.1 Substitution

The first phase in Tcl command processing is substitution. The Tcl interpreter scans commands from left to right. During this scan, it replaces phrases that should be substituted with the appropriate values. Tcl performs two types of substitutions:

- A Tcl command within square brackets (`[...]`) is replaced by the results of that command. This is referred to as command substitution.
- A variable preceded by a dollar sign is replaced by the value of that variable. This is referred to as variable substitution.

After these substitutions are done, the resulting command string is evaluated.

3.2.2 Controlling Substitutions with Quotes, Curly Braces, and the Backslash

Most Tcl commands expect a defined number of arguments and will generate an error if the wrong number of arguments is presented to them. When you need to pass an argument that consists of multiple words, you must group the words into a single argument with curly braces or with quotes.

The difference between grouping with quotes and grouping with braces is that substitutions will be performed on strings grouped with quotes but not on strings grouped with braces. Examples 6 and 7 show the difference between using quotes and curly braces.

The backslash may be used to disable the special meaning of the character that follows the backslash. You can escape characters such as the dollar sign, quote, or brace to disable their special meaning for Tcl. Examples 8 and 9 show the effects of escaping characters. A Tcl script can generate an error message with embedded quotes with code, as in the following.

```
puts "ERROR: Did not get expected \"+OK\" prompt"
```

The following examples show how quotes, braces, and backslashes affect the substitutions. The first example places the argument to `puts` within curly braces. No substitutions will occur.

Example 6

Script Example

```
set x 2
set y 3
puts {The sum of $x and $y is returned by [expr $x+$y]}
```

Script Output

```
The sum of $x and $y is returned by [expr $x+$y]
```

In Example 7, `puts` has its argument enclosed in quotes, so everything is substituted.

Example 7

Script Example

```
set x 2
set y 3
puts "The sum of $x and $y is [expr $x+$y]"
```

Script Output

```
The sum of 2 and 3 is 5
```

In Example 8, the argument is enclosed in quotes, so substitution occurs, but the square brackets are escaped with backslashes to prevent Tcl from performing a command substitution.

Example 8

Script Example

```
set x 2
set y 3
puts "The sum of $x and $y is returned by \[expr $x+$y\]"
```

Script Output

```
The sum of 2 and 3 is returned by [expr 2+3]
```

Example 9 escapes the dollar sign on the variables to prevent them from being substituted and also escapes a set of quotes around the `expr` string. If not for the backslashes before the quotes, the quoted string would end with the second quote symbol, which would be a syntax error. Sets of square brackets and curly braces nest, but quotes do not.

Example 9

Script Example

```
set x 2
set y 3
puts "The sum of \$x + \$y is returned by \"\[expr \$x+\$y\]\""
```

Script Output

```
The sum of $x + $y is returned by "[expr $x+$y]"
```

Splitting Lists

The {*} operator will convert a list to its component parts before evaluating a command. This is commonly used when one procedure returns a set of values that need to be passed to another procedure as separate values, instead of as a single list.

The `set` command requires two arguments to assign a value to a variable - the name of the variable and the value. You cannot assign the variable name and value to a string and then pass that string to the `set` command.

```
# This is an error
set nameANDvalue "a 2"
set $nameANDvalue
```

The {*} operator can split the string `"a 2"` into two components: the letter `a` and the number `2`.

Example 10

Script Example

```
# This works
set nameANDvalue "a 2"
set {*}$nameANDvalue
puts $a
```

Script Output

```
2
```

3.2.3 Steps in Command Evaluation

When a Tcl interpreter evaluates a command, it makes only one pass over that command to perform substitutions. It does not loop until a variable is fully resolved. However, if a command includes another Tcl command within brackets, the command processor will be called recursively until there are no further bracketed commands. When there are no more phrases to be substituted, the command is evaluated, and the result is passed to the previous level of recursion to substitute for the bracketed string.

The next example shows how the interpreter evaluates a command. The indentation depth represents the recursion level. Let's examine the following command.

```
set x [expr [set a 3] + 4 + $a]
```

The `expr` command performs a math operation on the supplied arguments and returns the results. For example, `expr 2+2` would return the value `4`.

The interpreter scans the command from left to right, looking for a phrase to evaluate and substitute. The scanner encounters the left square bracket, and the command evaluator is reentered with that subset of the command.

```
  expr [set a 3] + 4 + $a
```

The interpreter scans the new command, and again there is a bracket, so the command evaluator is called again with the following subset.

```
        set a 3
```

There are no more levels of brackets and no substitutions to perform, so this command is evaluated, the variable `a` is set to the value `3`, and `3` is returned. The recursive call returned `3`, so the value 3 replaces the bracketed command, and the command now resembles the following.

```
  expr 3 + 4 + $a
```

The variables are now substituted, and `$a` is replaced by `3`, making the following new command.

```
  expr 3 + 4 + 3
```

The interpreter evaluates this string, and the result (`10`) is returned. The substitution is performed, and the command is now as follows.

```
set x 10
```

The interpreter evaluates this string, the variable `x` is set to `10` , and `tclsh` returns `10`. In particular, note that the variable `a` was not defined when this command started but was defined within the first bracketed portion of the command. If this command had been written in another order, as in the following,

```
set x [expr $a + [set a 3] + 4 ]
```

the Tcl interpreter would attempt to substitute the value of the variable `a` before assigning the value `3` to `a`.

- If `a` had not been previously defined, it would generate an error.
- If `a` had been previously defined, the command would return an unexpected result depending on the value. For instance, if `a` contained an alphabetic string, `expr` would be unable to perform the arithmetic operation and would generate an error.

A Tcl variable can contain a string that is a Tcl command string. Dealing with these commands is discussed in Chapter 5.

3.3 DATA TYPES

The primitive data type in Tcl is the string (which may be a numeric value). The composite data types are the list, dict and associative array. The Tcl interpreter manipulates some complex entities such as graphic objects, I/O channels, and sockets via handles. Handles are introduced briefly here, with more discussion in the following chapters.

Unlike C, C++, or Java, Tcl is a typeless language. However, certain commands can define what sort of string data they will accept. Thus, the `expr` command, which performs math operations, will generate an error if you try to add 5 to the string “You can’t do that.”

3.3.1 Assigning Values to Variables

The command to define the value of a variable is `set`. It allocates space for a variable and data and assigns the data to that variable.

Syntax: `set` *`varName ?value?`*

Define the value of a variable.

`varName`	The name of the variable to define.
`value`	The data (value) to assign to the variable.

`set` always returns the value of the variable being referenced. When `set` is invoked with two arguments, the first argument is the variable name and the second is a value to assign to that variable. When `set` is invoked with a single argument, the argument is a variable name and the value of that variable is returned.

Example 11

```
% set x 1
1
% set x
1
% set z [set x 2]
2
% set z
```

```
2
%  set y
can't read "y": no such variable
```

Because Tcl is often used as a string processing language, it's also useful to be able to add new characters to the end of the value in a variable. The `append` command will append a string to the end of a variable. If the variable was not previously defined, the `append` command will create the variable and will assign the initial value to the string.

Syntax: `append` *`varName ?value1? ?value2?`*

Append one or more new values to a variable.

`varName`	The name of the variable to which to append the data.
`value`	The data to append to the variable content.

Note that `append` appends only the data you request. It does not add any separators between data values.

Example 12

```
%  set x 1
1
%  append x 2
12
%  append x
12
%  append x 3 4
1234
%  append y new value
newvalue
```

3.3.2 Strings

The Tcl interpreter represents all data as a string within a script. (Within the interpreter, the data may be represented in the computer's native format.) A Tcl string can contain alphanumeric, pure numeric, Boolean, or even binary data.

Alphanumeric data can include any letter, number, or punctuation. Tcl uses 16-bit Unicode to represent strings, which allows non-Latin characters (including Japanese, Chinese, and Korean) to be used in strings. A Tcl script can represent numeric data as integers, floating-point values (with a decimal point), hexadecimal or octal values, or scientific notation.

You can represent a Boolean value as a 1 (for true) and 0 (for false), or as the string `"true"` or `"yes"` and `"false"` or `"no"`. Any capitalization is allowed in the Boolean string: `"TrUe"` is recognized as a Boolean value. The command that receives a string will interpret the data as a numeric or alphabetic value, depending on the command's data requirements.

Example 13

Legitimate Strings

```
set alpha "abcdefg"
```

Assign the string "`abcdefg`" to the variable `alpha`.

```
set validString "this is a valid string"
```

Assign the string "`this is a valid string`" to the variable `validString`.

```
set number 1.234
```

Assign the number `1.234` to the variable `number`.

```
set octalVal 0755
```

Assign the octal value `755` to the variable `octalVal`. Commands that interpret values numerically will convert this value to 493 (base 10). Support for the leading 0 to represent octal numbers is still supported in Tcl 8.6, but may be removed in later releases.

```
set hexVal 0x1ed
```

Assign the hex value `1ED` to the variable `hexVal`. Commands that interpret values numerically will convert this value to 493 (base 10).

```
set scientificNotation 2e2
```

Assign the string `2e2` to the variable `scientificNotation`. Commands that interpret values numerically will convert this value to 200.

```
set msg {Bad input: "Bogus". Try again.}
```

Assign the string `Bad input: "Bogus". Try again.` to the variable `msg`. Note the internal quotes. Quotes within a braced string are treated as ordinary characters.

```
set msg "Bad input: \"Bogus\". Try again."
```

Assign the string `Bad input: "Bogus". Try again.` to the variable `msg`. Note that the internal quotes are escaped.

Bad Strings

```
set msg "Bad input: "Bogus". Try again."
```

The quotes around `Bogus` are not escaped and are treated as quotes. The quote before `Bogus` closes the string, and the rest of the line causes a syntax error.

```
Error Message: extra characters after close-quote
```

```
set badstring "abcdefg
```

Has only one quote. The error message for this will vary, depending on how the missing quote is finally resolved.

```
set mismatch {this is not a valid string"
```

Quote and brace mismatch. The error message for this will vary, depending on how the missing quote is finally resolved.

```
set noquote this is not a valid string
```

This set of words must be grouped to be assigned to a variable.

```
Error Message:  wrong # args: should be "set varName ?newValue?"
```

3.3.3 String Processing Commands

The `string`, `format`, and `scan` commands provide most of the tools a script writer needs for manipulating strings. The regular expression commands are discussed in Section 5.6. The `string` subcommands include commands for searching for substrings, identifying string matches, trimming unwanted characters, determining the contents of a string, replacing substrings and converting case. The `format` command generates formatted output from a format descriptor and a set of data (like the C library `sprintf` function). The `scan` command will extract data from a string and assign values to variables (like the C library `scanf` function).

All Tcl variables are represented as strings. You can use the string manipulation commands with integers and floating-point numbers as easily as with alphabetic strings. When a command refers to a position in a string, the character positions are numbered from 0, and the last position can be referred to as `end`.

There is more detail on all of the string subcommands in the Tcl reference and the companion website tutorials. The following subcommands are used in the examples in the next chapters.

The `string match` command searches a target string for a match to a pattern. The pattern is matched using the glob match rules. The rules for glob matching are as follows.

`*`	Matches 0 or more characters.
`?`	Matches a single character.
`[]`	Matches a character in the set defined within the brackets.
`[abc]`	Defines `abc` as the set.
`[m-y]`	Defines all characters alphabetically between `m` and `y` (inclusive) as the set.
`\?`	Matches the single character ?.

Note that the glob rules use `[ ]` in a different manner than the Tcl evaluation code. You must protect the brackets from `tclsh` evaluation, or `tclsh` will try to evaluate the phrase within the brackets as a command and will probably fail. Enclosing a glob expression in curly braces will accomplish this.

Syntax: `string match` *`pattern string`*

Returns 1 if *`pattern`* matches *`string`*, else returns 0.

`pattern`	The pattern to compare to *`string`*.
`string`	The string to match against the *`pattern`*.

Example 14

```
%  set str "This is a test, it is only a test"
This is a test, it is only a test
%  string match "*test*" $str
```

```
1
%  string match "not present" $str
0
```

The `string tolower` command converts a string to lowercase letters. Note that this is not done in place. A new string of lowercase letters is returned. The `string toupper` command converts strings to uppercase using the same syntax.

Syntax: `string tolower` *`string`*

Syntax: `string toupper` *`string`*

`string`	The string to convert.

Example 15

```
%  set upper [string toupper $str]
THIS IS A TEST, IT IS ONLY A TEST
%  set lower [string tolower $upper]
this is a test, it is only a test
```

The `string first` command returns the location of the first instance of a substring in a test string, or −1 if the pattern does not exist in the test string. The `string last` returns the character location of the last instance of the substring in the test string.

Syntax: `string first` *`substr string`*

Syntax: `string last` *`substr string`*

Return the location of the first (or last) occurrence of *`substr`* in *`string`*.

`substr`	The substring to search for.
`string`	The string to search in.

Example 16

```
%  set str "This is a test, it is only a test"
This is a test, it is only a test
%  set st_first [string first st $str]
12
%  set st_last [string last st $str]
31
```

The `string length` command returns the number of characters in a string. With Tcl 8.0 and newer, strings are represented internally as 2-byte Unicode characters. The value returned by `string length` is the number of characters, not bytes.

Syntax: `string length` *`string`*

Return the number of characters in *`string`*.

`string`	The string.

Example 17

```
set len [string length $str]
33
```

The `string range` command returns the characters between two points in the string.

Syntax: `string range` *`string first last`*

Returns the characters in *`string`* between *`first`* and *`last`*.

`string`	The string.
`first`	The position of the first character to return
`last`	The position of the last character to return

Example 18

```
% set subset [string range $str $st_first $st_last]
st, it is only a tes
```

The `string map` command replaces one or more substrings within a string with new values.

Syntax: `string map` *`map string`*

Return a modified *`string`* based on values in the *`map`*.

`map`	a set of string pairs that describe the changes to be made to the string. Each string pair is an *`old`* string and a *`new`* string which will replace *`old`* string.
`string`	A string to be modified.

Example 19

```
% string map {"a test" "an exam"} $str
This is an exam, it is only an exam
# The map may contain multiple pairs as
#    {old1 new1 old2 new2 ...}
% string map \
    {This These is are a {} test tests it they} $str
These are tests, they are only tests
```

The `string is` command will report what type of data is in a string. It can test to see if a string is an integer, double, printable string, boolean and more. See the man page for all the types of tests the `string is` command can perform.

Syntax: `string is` *`type string`*

Test to see if the string matches the type.

`type` The type of data that might be contained in string. Options include:

`digit`	Any unicode digit
`integer`	An integer value. May include leading or trailing whitespace.
`double`	A number
`alnum`	A letter or number
`upper`	Uppercase letters
`lower`	Lowercase letters
`space`	Any unicode space character

Example 20

```
% string is integer "123"
1
% string is integer "123a"
0
% string is integer "123.0"
0
% string is double "123.0"
1
% string is alnum "123a"
1
% string is alnum "123.0a"
0
```

The `format` command generates formatted strings and can perform some data conversions. It is equivalent to the C language `sprintf` command.

Syntax: `format` *`formatString ?data1? ?data2? ...`*

Return a new formatted string.

`formatString`	A string that defines the format of the string being returned.
`data#`	Data to substitute into the formatted string.

The first argument must be a format description. The format description can contain text strings and % fields. The text string will be returned exactly as it appears in the format description. The % fields will be substituted with formatted strings derived from the data that follows the format descriptor. A literal percent symbol can be generated with a %% field.

The format for the % fields is the same as that used in the C library. The field definition is a string consisting of a leading percent sign, two optional fields, and a '`formatDefinition`, as follows.

`% ?justification? ?field width? formatDefinition`

- The first character in a % field is the `%` symbol.
- The *`field justification`* may be a plus or minus sign. A minus sign causes the content of the % field to be left justified. A plus sign causes the content to be right justified. By default the data is right justified.
- The *`field width`* is a numeric field. If it is a single integer, it defines the width of the field in characters. If this value is two integers separated by a decimal point, the first integer represents the total width of the field in characters, and the second represents the number of digits to the right of the decimal point to display for floating-point formats.
- The *`formatDefinition`* is the last character. It must be one of the following.

s The argument should be a string.

Replace the field with the argument.

```
% format %s 0xf
0xf
```

c The argument should be a decimal integer.

Replace the field with the ASCII character value of this integer.

```
% format %c 65
A
```

d or i The argument should be a decimal integer.

Replace the field with the decimal representation of this integer.

```
% format %d 0xff
255
```

u The argument should be an integer.

Replace the field with the decimal representation of this integer
treated as an unsigned value.

```
% format %u -1
4294967295
```

o The argument should be an decimal integer value.

Replace the field with the octal representation of the argument.

```
% format %o 0xf
17
```

X or x The argument should be a decimal integer.

Replace the field with the hexadecimal representation of this integer.

```
% format %x -1
ffffffff
```

f The argument should be a numeric value.
Replace the field with the decimal fraction representation.

```
% format %3.2f 1.234
1.23
```

E or e The argument should be a numeric value.
Replace the field with the scientific notation representation of this integer.

```
% format %e 0xff
2.550000e+02
```

G or g The argument should be a numeric value.
Replace the field with the scientific notation or floating-point representation.

```
% format %g 1.234e2
123.4
```

Example 21

```
% format {%5.3f} [expr 2.0/3]
0.667
% format {%c%c%c%c%c} 65 83 67 73 73
ASCII
% set def "%-12s %s"
% puts [format $def "Author" "Title"]
% puts [format $def "Clif Flynt" \
    "Tcl/Tk: A Developers Guide"]

Author       Title
Clif Flynt   Tcl/Tk: A Developers Guide
```

The `scan` command is the flip side to `format`. Instead of formatting output, the `scan` command will parse a string according to a format specifier. The `scan` command emulates the behavior of the C `sscanf` function. The first argument must be a string to scan. The next argument is a format description, and the following arguments are a set of variables to receive the data values.

Syntax: `scan` *`textString formatString ?varName1? ... ?varNameN?`*

Parse a text string into one or more variables.

textString	The text data to scan for values.
formatString	Describes the expected format for the data. The format descriptors are the same as for the `format` command.
*varName**	The names of variables to receive the data.

The `scan` command returns the number of percent fields that were matched. If this is not the number of percent fields in the `formatString`, it indicates that the `scan` command failed to parse the data.

The format string of the `scan` command uses the same `%` descriptors as the `format` command, and adds a few more.

[...] The value between the open and close square brackets will be a list of characters that can be accepted as matches.

Characters can be listed as a range of characters ([a-z]). A leading or trailing dash is considered a character, not a range marker.

All characters that match these values will be accepted until a nonmatching character is encountered.

```
% scan "a scan test" {%[a-z]} firstword
1

% set firstword

a
```

[^...] The characters after the caret (^) will be characters that cannot be accepted as matches. All characters that do not match these values will be accepted until a matching character is encountered.

```
% scan "a scan test" {%[^t-z]} val
1
% set val
a scan
```

In the following example, the format string {`%s %s %s %s`} will match four sets of non-whitespace characters (words) separated by whitespace.

Example 22

```
%  set string {Speak, Friend and Enter}
Speak, Friend and Enter
% scan $string {%s %s %s %s} a b c d
4
% puts "The Password is: $b"
The Password is: Friend
```

A format string can also include literal characters that will be included in a `format` return, or must be matched by the `scan` command. For instance, the `scan` command in the previous example could also be written as follows.

```
% scan $string {Speak %s} password
```

This would extract the password from `Speak Friend and Enter`, but would not extract any words from "`The password is sesame`", since the format string requires the word *Speak* to be the first word in the string.

String and Format Command Examples

This example shows how you might use some `string, scan`, and `format` commands to extract the size, from, and timestamp data from an e-mail log file entry and generate a formatted report line.

Example 23

Script Example

```
# Define the string to parse.
set logEntry {Mar 25 14:52:50 clif sendmail[23755]:
from=<tcl-core-admin@lists.sourceforge.net>,
size=35362, class=-60, nrcpts=1

# Extract "From" using string first and string range
set openAnglePos [string first "<" $logEntry]
set closeAnglePos [string first ">" $logEntry]
set fromField [string range $logEntry $openAnglePos $closeAnglePos]

# Extract the date using scan
scan $logEntry {%s %d %d:%d} mon day hour minute
# Extract the size using scan and string cmds.
set sizeStart [string first "size=" $logEntry]
set substring [string range $logEntry $sizeStart end]

# The formatString looks for a word composed of the
# letters ‘eisz’ (size will match) followed by an
# equals sign, followed by an integer. The word
# ‘size’ gets placed in the variable discard,
# and the numeric value is placed in the variable
# sizeField.
scan $substring {%[eisz]=%d} discard sizeField


puts [format {%-12s %-40s %-s} "Timestamp" "From" "Size"]
puts [format {%s %d %d:%d %-40s %d} \
    $mon $day $hour $minute $fromField $sizeField]
```

Script Output

```
Timestamp    From                                     Size
Mar 25 14:52 <tcl—core—admin@lists.sourceforge.net> 35362
```

3.3.4 Lists

A Tcl list can be represented as a string that follows some syntactic conventions. (Internally, a string is represented as a list of pointers to Tcl objects, which are discussed later.)

- A list can be represented as a list of list elements enclosed within curly braces.
- Each word is a list element.
- A set of words may be grouped with curly braces.
- A set of words grouped with curly braces is a list element within the larger list, and also a list in its own right.
- A list element can be empty (it will be displayed as {}).

For example, the string {apple pear banana} can be treated as a list. The first element of this list is apple, the second element is pear, and so on. The order of the elements can be changed with Tcl commands for inserting and deleting list elements, but the Tcl interpreter will not modify the order of list elements as a side effect of another operation.

A list may be arbitrarily long, and list elements may be arbitrarily long. Any string that adheres to these conventions can be treated as a list, but it is not guaranteed that any arbitrary string is a valid list. For example, "this is invalid because of an unmatched brace {" is not a valid list.

With Tcl 8.0, lists and strings are treated differently within the interpreter. If you are dealing with data as a list, it is more efficient to use the list commands. If you are dealing with data as a string, it is better to use the string commands.

The following are valid lists.

```
{This is a six element list}
{This list has {a sublist} in it}
{Lists may {be nested {arbitrarily deep}}}
"A string like this may be treated as a list"
```

The following are invalid lists.

```
{This list has mismatched braces
{This list {also has mismatched braces
```

3.3.5 List Processing Commands

A list can be created in the following ways.

- By using the set command to assign a list to a variable
- By grouping several arguments into a single list element with the list command
- By appending data to an unused variable with the lappend command
- By splitting a single argument into list elements with the split command
- By using a command that returns a list. (The array names command returns a list of associative array indices. It will be discussed in Section 3.3.7.)

The list command takes several units of data and combines them into a single list. It adds whatever braces may be necessary to keep the list members separate.

Syntax: list *element1 ?element2? ... ?elementN?*

Creates a list in which each argument is a list element.

*element** A unit of data to become part of the list

Example 24

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
```

The lappend command appends new data to a list, creating and returning a new, longer list. Note that this command will modify the existing list, unlike the string commands, which return new data without changing the original.

Syntax: `lappend` *`listName ?element1? ... ?elementN?`*

Appends the arguments onto a list.

`listName`	The name of the list to append data to.
`element*`	A unit of data to add to the list.

Example 25

```
% lappend mylist fifth
first second {three element sublist} fourth fifth
```

The `split` command returns the input string as a list. It splits the string wherever certain characters appear. By default, the split location is a `whitespace` character: a space, tab, or newline.

Syntax: `split` *`data ?splitChar?`*

Split data into a list.

`data`	The string data to split into a list.
`?splitChar?`	An optional character (or list of characters) at which to split the data.

Example 26

```
% set commaString "1,2.2,test"
1,2.2,test
% # Split on commas
% set lst2 [split $commaString ,]
1 2.2 test
% # Split on comma or period
% set lst2 [split $commaString {,.}]
1 2 2 test
% # Split on empty space between letters
% # (each character becomes a list element)
% set lst2 [split $commaString {}]
1 , 2 . 2 , t e s t
```

Tcl also includes several commands for manipulating lists. These include commands to convert a list into a string, return the number of elements in a list, search a list for elements that match a pattern, retrieve particular elements from a list, and insert and replace elements in a list.

The `join` command converts a list into a string.

Syntax: `join` *`list ?separator?`*

Joins the elements of a list into a string.

`list`	The list to convert to a string.
`separator`	An optional string that will be used to separate the list elements. By default, this is a space.

The `join` command can be used to convert a Tcl list into a comma-delimited list for import into a spreadsheet.

Example 27

```
% set numbers [list 1 2 3 4]
1 2 3 4
% join $numbers :
1:2:3:4
% join $numbers ", "
1, 2, 3, 4
```

The `llength` command returns the number of list elements in a list. Note that this is not the number of characters in a list, but the number of list elements. List elements may be lists themselves. These lists within a list are each counted as a single list element.

Syntax: `llength` *`list`*

Returns the length of a list.

`list`	The list.

Example 28

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% llength $mylist
4
```

The `lsearch` command will search a list for elements that match a pattern. The `lsearch` command uses the glob-matching rules by default. These are described with the previous `string match` discussion. The regular expression rules are discussed in Chapter 5.

Syntax: `lsearch` *`?-option? list pattern`*

Returns the index of the first list element that matches *`pattern`* or `-1` if no element matches the pattern. The first element of a list has an index of 0.

`?-option?` Controls how the `lsearch` command will behave. Multiple options may be used together to control the behavior of the `lsearch` command. Some of the modifiers include:

`-exact`	List element must exactly match the pattern.
`-glob`	List element must match pattern using the `glob` rules. This is the default matching algorithm.
`-regexp`	List element must match pattern using the regular expression rules.

`-all`	Return all the values that match the pattern, instead of only the first element.
`-inline`	Return the element instead of the index of the element.
`-start` *`position`*	Return the first value after *`position`* that matches the pattern.
`-not`	Inverts the test and returns values that do not match the pattern.

`list`	The list to search.
`pattern`	The pattern to search for.

Example 29

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% lsearch $mylist second
1
% # three is not a list element - it's a part of a list element
% lsearch $mylist three
-1
% lsearch $mylist "three*"
2
% lsearch $mylist "*ou*"
3
% lsearch -all $mylist "*s*"
0 1 2
% lsearch -all -inline $mylist "*s*"
first second {three element sublist}
% lsearch -start 1 -all -inline $mylist "*s*"
second {three element sublist}
% lsearch -not $mylist "*s*"
3
```

You can extract elements from a list with the `lindex` command.

Syntax: `lindex` *`list index`*

Returns a list element. The first element is element 0. If this value is larger than the list, an empty string is returned.

`list`	The list.
`index`	The position of a list entry to return.

Example 30

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
```

```
% lindex $mylist 0
first
% lindex $mylist 2
three element sublist
% lindex $mylist [lsearch $mylist *ou*]
fourth
```

The `linsert` command returns a new list with the new elements inserted. It does not modify the existing list.

Syntax: `linsert` *`list position element1 ... ?elementN?`*

Inserts an element into a list at a given position.

`list`	The list to receive new elements.
`position`	The position in the list at which to insert the new list elements. If this value is `end` or greater than the number of elements in the list, the new values are added at the end of the list.
`element*`	One or more elements to be inserted into the list.

Example 31

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% set longerlist [linsert $mylist 0 zero]
zero first second {three element sublist} fourth
% puts $mylist
first second {three element sublist} fourth
```

Like `linsert`, the `lreplace` command returns a new list. It does not modify the existing list. The difference between the *`first`* and *`last`* elements need not match the number of elements to be inserted. This allows the `lreplace` command to be used to increase or decrease the length of a list.

Syntax: `lreplace` *`list first last element1 ... ?elementN?`*

Replaces one or more list elements with new elements.

`list`	The list to have data replaced.
`first`	The first position in the list at which to replace elements. If this value is `end`, the last element will be replaced. If the value is greater than the number of elements in the list, an error is generated.
`last`	The last element to be replaced.
`element*`	Zero or more elements to replace the elements between the *`first`* and *`last`* elements.

Example 32

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% set newlist [lreplace $mylist 0 0 one]
one second {three element sublist} fourth
% set shortlist [lreplace $mylist 0 1]
{three element sublist} fourth
```

The next example demonstrates using the list commands to split a set of comma and newline delimited data (a common export format for spreadsheet programs) into a Tcl list and then reformat the data for display.

Example 33

List Commands Example

```
# Define the raw data

set rawData {Package,Major,Minor,Patch
Tcl,8,6,0
math::geometry,1,0,3
math::complexnumbers,1,0,2}

# Split the raw data into a list using the newlines
#     as list element separators.
# This creates a list in which each line becomes a
#    list element
set dataList [split $rawData "\n"]

# Create a list of the column headers.
set columns [split [lindex $dataList 0] ","]

foreach line [lrange $dataList 1 end] {
  # Convert the line of data into a list
  set rowValues [split $line ","]\

  # Create a new list from $rowValues that includes
  # all the elements after the first (package name).
  set revList [lrange $rowValues 1 end]

  # and rejoin them into a "." separated string
  set revision [join $revList "."]

  # Display a reformatted version of the data line
```

```
    puts [format "%s: %s Revision: %s" [lindex $columns 0]\
        [lindex $rowValues 0] $revision]
  }
```

Script Output

```
Package: Tcl Revision: 8.6.0
Package: math::geometry Revision: 1.0.3
Package: math::complexnumbers Revision: 1.0.2
```

3.3.6 Dictionaries

The `dict` command was added to Tcl in version 8.5. Conceptually, a dictionary is an ordered list of key-value pairs. A dict looks like this:

```
puts [dict create key1 val1 key2 val2]

key1 val1 key2 val2
```

The concept of key-value pairs is simple enough that you may be tempted to just write procedures to handle such lists. Writing procedures to handle lists of key-value pairs will be demonstrated in Chapter 6.

Unless you are constrained to use versions of Tcl that don't have dictionary support, you should use the `dict` command. The improvements of the dictionary over home-grown keyed-list procedures are:

- the dict supports nesting dictionaries within a dictionary (just as lists can be nested).
- the `dict` command is implemented in fast C code with hash tables.
- there is a rich set of `dict` subcommands to search and manipulate dictionaries.
- the `dict` command contains both field and value information, making it useful as a data construct to use with a database extension.

As with Tcl lists, a dict variable can be initialized in several ways including `dict create` , `dict append` and `dict lappend`.

The `dict create` command creates a complete dictionary without assigning the value to a variable, just as you might use the `format` command to initialize a string.

Syntax: `dict create` *`key1 val1 key2 val2 ...`*

Create a dictionary of keys and values.

`key*`	A key in the dictionary.
`val*`	The value to associate with the previous key.

The next example creates a dictionary of movie titles and notable quotes. If you print the contents of the `$quotes` variable it will look just like a list of movie titles and quotes.

Example 34

Creating a Dict

```
set quotes [dict create \
  "Casablanca" "Play it, Sam." \
  "Star Wars"  "I get a bad feeling about this." \
  "Indiana Jones" "Snakes, Why did it have to be snakes." \
  "Looney Tunes" "What's Up Doc?"]

set movie Casablanca
puts "A notable quote from $movie is: [dict get $quotes $movie]"
```

Script Output

```
A notable quote from Casablanca is: Play it, Sam.
```

In the previous example, the values associated with each key are a grouped set of words, but there is no higher level of grouping. If we want each quote to be a single unit (so we can have multiple quotes for some movies), the `dict create` command will need to add the grouping as shown in the next example.

Example 35

Creating a Dict of Quotes

```
set quotes [dict create \
  "Casablanca" [list "Play it, Sam." "Round up the usual suspects"] \
  "Star Wars"  [list "I get a bad feeling about this."] \
  "Indiana Jones" [list "Snakes, Why did it have to be snakes."]]

set movie Casablanca
puts "A notable quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
A notable quote from Casablanca is: Play it, Sam.
```

The `dict append` and `dict lappend` commands will modify an existing dict variable, or create a new variable if the variable did not previously exist. This is the same way that the `append` and `lappend` commands work with string and list variables.

Syntax: `dict append` *`dictName key value`*

Appends the given value to the given key.

`dictName`	Name of the variable to be modified.
`key`	Key within this dictionary to be modified.
`value`	Value to append onto the current value of this key.

Syntax: `dict lappend` *`dictName key value`*

Appends the given value to the given key.

`dictName`	Name of the variable to be modified.
`key`	Key within this dictionary to be modified.
`value`	Value to lappend onto the current value of this key.

Example 36

Modifying a Dict

```
# Add a new quote to the Star Wars quotes
dict lappend quotes "Star Wars" "Feel the Force, Luke."

# Create a new entry for ET quotes.
dict lappend quotes "ET" "E.T. Phone Home."

set movie "Star Wars"
puts "Notable quotes from $movie include:"
foreach quote [dict get $quotes $movie] {
  puts "  $quote"
}
```

Script Output

```
Notable quotes from Star Wars include:
I get a bad feeling about this.
Feel the Force, Luke.
```

Because the values in this dictionary are lists, modifying an individual list element must be done by extracting the value from the dictionary, modifying it and replacing it. For example, the complete *Casablanca* quote is “Play it, Sam. Play *As Time Goes By.*”

The `dict replace` command will let us replace a value with a new value. Like the `lreplace` command, this command does not replace the value in the dictionary, it returns a new dictionary with the value modified.

Syntax: `dict replace` *`$dict key new_value`*

Return a new dictionary with the value associated with a key replaced with a new value.

`$dict`	The dictionary to be modified.
`key`	The key to be modified.
`new_value`	The new value for this key.

Example 37

Modifying a Dict Value

```
set vals [dict get $quotes Casablanca]
set val [lindex $vals 0]
append val "  Play ‘As Time Goes By’."
set vals [lreplace $vals 0 0 $val]
set quotes [dict replace $quotes Casablanca $vals]

set movie Casablanca
puts "The full quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
The full quote from Casablanca is:
Play it, Sam. Play ‘As Time Goes By’.
```

A dictionary value can be modified in place with the `dict set` command.

Syntax: `dict set` *`dictName key new_value`*

Sets the contents of a dictionary key to a new value.

$dict	The name of the dictionary to be modified.
key	The key to be modified.
new_value	The new value for this key.

The previous example—adding a string to one movie quote—becomes a bit simpler by using the `dict set` command instead of `dict replace`.

Example 38

Modify a Dict Value

```
set vals [dict get $quotes Casablanca]
set val [lindex $vals 0]
append val "  Play ‘As Time Goes By’."
set vals [lreplace $vals 0 0 $val]
dict set quotes Casablanca $vals

set movie Casablanca
puts "The full quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
The full quote from Casablanca is:
Play it, Sam. Play ‘As Time Goes By’.
```

3.3.7 Associative Arrays

The associative array is an array that uses a string to index the array elements, instead of a numeric index the way C, FORTRAN, and BASIC implement arrays. A variable is denoted as an associative array by placing an index within parentheses after the variable name.

For example, `price(apple)` and `price(pear)` would be associative array variables that could contain the price of an apple or pear. The associative array is a powerful construct in its own right and can be used to implement composite data types resembling the C struct or even a C++ class object. Using associative arrays is further explored in Chapter 6.

Example 39

`set price(apple) .10`	`price` is an associative array. The element referenced by the index `apple` is set to .10.
`set price(pear) .15`	`price` is an associative array. The element referenced by the index `pear` is set to .15.
`set quantity(apple) 20`	`quantity` is an associative array. The element referenced by the index `apple` is set to 20.
`set discount(12) 0.95`	`discount` is an associative array. The element referenced by the index `12` is set to 0.95.

3.3.8 Associative Array Commands

An array element can be treated as a simple Tcl variable. It can contain a number, string, list, or even the name of another array element. As with lists, there is a set of commands for manipulating associative arrays. You can get a list of the array indices, get a list of array indices and values, or assign many array indices and values in a single command. Like the `string` commands, the `array` commands are arranged as a set of subcommands of the `array` command.

The list of indices returned by the `array names` command can be used to iterate through the content of an array.

Syntax: `array names` *`arrayName ?pattern?`*

Returns a list of the indices used in this array.

`arrayName`	The name of the array.
`pattern`	If this option is present, `array names` will return only indices that match the pattern. Otherwise, `array names` returns all the array indices.

Example 40

```
set fruit(apples) 10
set fruit(pears) 5
foreach item [array names fruit *] {
  puts "There are $fruit($item) $item."
}
```

```
There are 5 pears.
There are 10 apples.
```

The `array get` command returns the array indices and associated values as a list. The list is a set of pairs in which the first item is an array index and the second is the associated value. The third list element will be another array index (first item in the next pair), the fourth will be the value associated with this index, and so on.

Syntax: `array get` *`arrayName`*

Returns a list of the indices and values used in this associative array.

`arrayName`	The name of the array.

Example 41

```
% array get fruit
pears 5 apples 10
```

The `array set` command accepts a list of values in the format that `array get` generates. The `array get` and `array set` pair of commands can be used to copy one array to another, save and restore arrays from files, and so on.

Syntax: `array set` *`arrayName`* `{`*`index1 value1 ... indexN valueN`*`}`

Assigns each value to the appropriate array index.

`arrayName`	The name of the array.
`index*`	An index in the array to assign a value to.
`value*`	The value to assign to an array index.

Example 42

```
% array set fruit [list bananas 20 peaches 40]
% array get fruit
bananas 20 pears 5 peaches 40 apples 10
```

The next example shows some simple uses of an array. Note that while the Tcl array does not explicitly support multiple dimensions the index is a string and you can define multidimensional arrays by using a naming convention, such as separating fields with a comma, period, dash, and so on, that does not otherwise appear in the index values.

Example 43

Array Example

```
# Initialize some values with set
set fruit(apple.cost) .10
set fruit(apple.count) 5
```

```
# Initialize some more with array set
array set fruit {pear.cost .15 pear.count 3}
# At this point the array contains:

# Index        Value
# apple.cost   .10
# pear.cost    .15
# apple.count  5
# pear.count   3

# Count the number of different types of fruit in the
# array by getting a list of unique indices, and then
# using the llength command to count the number of
# elements in the list.
set typeCount [llength [array names fruit *cost]]
puts "There are $typeCount types of fruit in the fruit array"
# You can use another variable to hold all,
# or a part of an array index.
set type apple
puts "There are $fruit($type.count) apples"
set type pear
puts "There are $fruit($type.count) pears"
array set new [array get fruit]
set type pear
puts "There are $new($type.count) pears in the new array"
```

Script Output

```
There are 2 types of fruit in the fruit array
There are 5 apples
There are 3 pears
There are 3 pears in the new array
```

3.3.9 Binary Data

Versions of Tcl prior to 8.0 (pre 1998) used null-terminated ASCII strings for the internal data representation. This made it impossible to use Tcl with binary data that might have NULLs embedded in the data stream.

With version 8.0, Tcl moved to a new internal data representation that uses a native-mode data representation. An integer value is saved as a long integer, a real value is saved as an IEEE floating point value, and so on. The new method of data representation supports binary data, and a command was added to convert binary data to integers, floats, or strings. Tcl is still oriented around printable ASCII strings but the binary command makes it possible to handle binary data easily.

The `binary` command supports two subcommands to convert data to and from a binary representation. The `format` subcommand will transform an ASCII string to a binary value, and the `scan` subcommand will convert a string of binary data to one or more printable Tcl variables. These subcommands require a descriptor to define the format of the binary data. Examples of these descriptors follow the command syntax.

Syntax: `binary format` *`format arg1 ?arg2? ... ?argN?`*

Returns a binary string created by converting one or more printable ASCII strings to binary format.

`format`	A string that describes the format of the ASCII data.
`arg*`	The printable ASCII to convert.

Syntax: `binary scan` *`binaryData format arg1 ?varName1? ... ?varNameN?`*

Converts a string of binary data to one or more printable ASCII strings.

`binaryData`	The binary data.
`format`	A string that describes the format of the ASCII data.
`varName*`	Names of variables to accept the printable representation of the binary data.

The components of *`format`* are similar to the format strings used by `scan` and `format` in that they consist of a descriptor (a letter) and an optional count. If the count is defined, it describes the number of items of the previous type to convert. The count defaults to 1. Common descriptors include the following.

Example 44

Script Example

```
binary scan "Tk" H4 x
puts "X: $x"
# Assign three integer values to variables
set a 1415801888
set b 1769152615
set c 1919246708
# Convert the integers to ASCII equivalent
puts [binary format {I I2} $a [list $b $c]]
```

Script Output

```
X: 546b
Tcl is great
```

The string used to define the format of the binary data is very powerful, and allows a script to extract fields from complex C structures.

h Converts between binary and hexadecimal digits in little endian order.

`binary format h2 34` - returns “C” (0x43).

`binary scan "4" h2 x` - stores 0x43 in the variable x

H Converts between binary and hexadecimal digits in big endian order.

`binary format H2 34` - returns “4” (0x34).

`binary scan "4" H2 x` - stores 0x34 in the variable x c Converts an 8-bit value to/from ASCII.

`binary format c 0x34` - returns “4” (0x34).

`binary scan "4" c x` - stores 0x34 in the variable x

s Converts a 16-bit value to/from ASCII in little endian order.

`binary format s 0x3435` - returns “54” (0x35 0x34).

`binary scan "45" s x` - stores 13620 (0x3534) in the variable x

S Converts a 16-bit value to/from ASCII in big endian order.

`binary format S 0x3435` - returns ”45” (0x34 0x35).

`binary scan "45" S x` - stores 13365 (0x3435) in the variable x

i Converts a 32-bit value to/from ASCII in little endian order.

`binary format i 0x34353637` - returns ”7654” (0x37 0x36 0x35 0x34).

`binary scan "7654" ix` - stores 875902519 (0x34353637) in the variable x

I Converts a 32-bit value to/from ASCII in big endian order.

`binary format I 0x34353637` - returns ”4567” (0x34 0x35 0x36 0x37).

`binary scan "7654" Ix` - stores 926299444 (0x37363534) in the variable x

f Converts 32-bit floating point values to/from ASCII.

`binary format f 1.0` - returns the binary string ”0x00803f”.

`binary scan "0x00803f" f x` - stores 1.0 in the variable x

Example 45

Script Examples

```
C Code to Generate a Structure   Tcl Code to Read the Structure
#include <stdio.h>                # Open the input file, and read data
#include <fcntl.h>                set if [open tstStruct r]
main () {                         set d [read $if]
  struct a {                      close $if
    int i;
    float f[2];                   # scan the binary data into variables.
    char s[20];
    } aa;                         binary scan $d "i f2 a*" i f s
    FILE *of;                     # The string data includes any binary
                                  # garbage after the NULL byte.
    aa.i = 100;                   # Strip off that junk.
    aa.f[0] = 2.5;
    aa.f[1] = 3.8;                set 0pos [string first\
                                    [binary format c 0x00] $s]
    strcpy(aa.s, "This is a test"); incr 0pos -1
                                  set s [string range $s 0 $0pos]
```

```
        of = fopen("tstStruct", "w");
        fwrite(sizeof(aa), 1, of);    # Display the results
        fclose(of);                   puts $i
    }                                 puts $f
                                      puts $s
```

Script Output

```
100
2.5 3.79999995232
This is a test
```

3.3.10 Handles

Tcl uses handles to refer to certain special-purpose objects. These handles are returned by the Tcl command that creates the object and can be used to access and manipulate the object. When you open a file, a handle is returned for accessing that file. The graphic objects created by a `wish` script are also accessed via handles, which will be discussed in the `wish` tutorial. The following are types of handles.

`channel`	A handle that references an I/O device such as a file, serial port, or TCP socket. A `channel` is returned by an `open` or `socket` call and can be an argument to a `puts, read, close, flush,` or `gets` call.
`graphic`	A handle that refers to a graphic object created by a `wish` command. This handle is used to modify or query an object.
`http`	A handle that references data returned by an `http::geturl` operation. An `http` handle can be used to access the data that was returned from the `http::geturl` command or otherwise manipulate the data.

There will be detailed discussion of the commands to manipulate handles in sections that discuss that type of handle.

3.4 ARITHMETIC AND BOOLEAN OPERATIONS

The commands discussed so far directly manipulate particular types of data. Tcl also has a rich set of commands for performing arithmetic and Boolean operations and for using the results of those operations to control program flow.

3.4.1 Math Operations

Math operations are performed using the `expr` and `incr` commands. The `expr` command provides an interface to a general-purpose calculation engine, and the `incr` command provides a fast method of changing the value of an integer.

The `expr` command will perform arbitrarily complex math operations. Unlike most Tcl commands, `expr` does not expect a fixed number of arguments. It can be invoked with the arguments grouped as a string or as individual values and operators. The `expr` command is optimized to handle arithmetic strings. You will see a performance improvement if you pass the `expr` command a string enclosed in curly braces, rather than using quotes which pushes some processing into the Tcl interpreter, instead of the `expr` command code.

The arithmetic arguments to `expr` may be grouped with parentheses to control the order of math operations. The `expr` command can also evaluate Boolean expressions and is used to test conditions by the Tcl branching and looping commands.

Syntax: `expr` *`mathExpression`*

Tcl supports the following math operations (grouped in decreasing order of precedence).

− + ~ !	Unary minus, unary plus, bitwise NOT, logical NOT.
* / %	Multiply, divide, modulo (return the remainder).
+ −	Add, subtract.
≪ ≫	Left shift, right shift.
< > <= >=	Less than, greater than, less than or equal, greater than or equal.
== !=	Equality, inequality.
&	Bitwise AND.
^	Bitwise exclusive OR.
|	Bitwise OR.
&&	Logical AND. Produces a 1 result if both operands are nonzero; 0 otherwise.
||	Logical OR. Produces a 0 result if both operands are zero; 1 otherwise.
x?y:z	If-then-else, as in C. If *x* evaluates to nonzero, the result is the value of *y*. Otherwise, the result is the value of *z*. The *x* operand must have a numeric value. The *y* and *z* operands may be variables or Tcl commands.

Note that the bitwise operations are valid only if the arguments are integers (not floating-point or scientific notation). The `expr` command also supports the following math functions and conversions.

Trigonometric Functions

sin	`sin(`*`radians`*`)`
	`set sin [expr sin($degrees/57.32)]`
cosine	`cos (`*`radians`*`)`
	`set cosine [expr cos(3.14/2)]`
tangent	`tan (`*`radians`*`)`
	`set tan [expr tan($degrees/57.32)]`
arcsin	`asin (`*`float`*`)`
	`set angle [expr asin(.7071)]`

arccosine	`acos` (*float*)
	`set angle [expr acos(.7071)]`
arctangent	`atan` (*float*)
	`set angle [expr atan(.7071)]`
hyperbolic sin	`sinh` (*radians*)
	`set hyp_sin [expr cosh(3.14/2)]`
hyperbolic cosine	`cosh` (*radians*)
	`set hyp_cos [expr cosh(3.14/2)]`
hyperbolic tangent	`tanh` (*radians*)
	`set hyp_tan [expr tanh(3.14/2)]`
hypotenuse	`hypot` (*float, float*)
	`set len [expr hypot($side1, $side2)]`
arctangent of ratio	`atan2` (*float, float*)
	`set radians [expr atan2($numerator, $denom)]`

Exponential Functions

natural log	`log` (*float*)
	`set two [expr log(7.389)]`
log base 10	`log10` (*float*)
	`set two [expr log10(100)]`
square root	`sqrt` (*float*)
	`set two [expr sqrt(4)]`
exponential	`exp` (*float*)
	`set seven [expr exp(1.946)]`
power	`pow` (*float, float*)
	`set eight [expr pow(2, 3)]`

Conversion Functions

Return closest int	`round` (*float*)
	`set duration [expr round($distance / $speed)]`
Largest integer less than a float	`floor` (*float*)
	`set overpay [expr ceil($cost / $count)]`
Smallest integer greater than a float	`ceil` (*float*)
	`set each [expr ceil($cost/$count)]`
Floating Point remainder	`fmod` (*float, float*)
	`set missing [expr fmod($cost, $each)]`

Convert int to float	`double` (*int*)
	`set average [expr $total / double($count)]`
Convert float to int	`int` (*float*),
	`set leastTen [expr (int($total) / 10) * 10]`
Absolute Value	`abs` (*num*)
	`set xDistance [expr abs($x1 - $x2)]`

Random Numbers

Seed random number	`srand` (*int*)
	`expr srand([clock seconds])`
Generate random number	`rand()`
	`set randomFloat [expr rand() ]`

Example 46

```
% set card [expr rand()]
0.557692307692
% set cardNum [expr int($card * 52)]
29
% set cardSuit [expr int($cardNum / 13)]
2
% set cardValue [expr int($cardNum % 13)]
3
% expr floor(sin(3.14/2) * 10)
9.0
% set x [expr int(rand() * 10)]
4
% expr atan(((3 + $x) * $x)/100.)
0.273008703087
```

The `incr` command provides a shortcut to modify the content of a variable that contains an integer value. The `incr` command adds a value to the current content of a given variable. The value may be positive or negative, thus allowing the `incr` command to perform a decrement operation. The `incr` command is used primarily to adjust loop variables.

Syntax: `incr` *`varName ?incrValue?`*

`incr`	Add a value (default 1) to a variable.
`varName`	The name of the variable to increment. NOTE: This is a variable name, not a value. Do not start the name with a `$`. This variable must contain an integer value, not a floating-point value.
`?incrValue?`	The value to increment the variable by. May be a positive or negative number. The value must be an integer between −65,536 and 65,535, not a floating-point value.

Example 47

```
% set x 4
4
% incr x
5
% incr x -3
2
% set y [incr x]
3
% puts "x: $x y: $y"
x: 3 y: 3
```

3.4.2 Conditionals

The `if` Command

Tcl supports both a single-choice conditional (`if`) and a multiple-choice conditional (`switch`). The `if` command tests a condition, and if that condition is true, the script associated with this test is evaluated. If the condition is not true, an alternate choice is considered, or alternate script is evaluated.

Syntax:
```
if {testExpression1} {
    body1
    } ?elseif {testExpression2} {
    body2
    }? ?else {
    bodyN
    }?
```

`if`	Determine whether a code body should be evaluated based on the results of a test. If the test returns true, the first body is evaluated. If the test is false and a body of code exists after the `else`, that code will be evaluated.
`testExpression1`	If this expression evaluates to true, the first body of code is evaluated. The expression must be in a form acceptable to the `expr` command. These forms include the following.

An arithmetic comparison

`{$a < 2}.`

A string comparison

`{ $string` *ne* `"OK"}.`

The results of a command

`{ [eof $inputFile]}.`

	A variable with a numeric value. Zero (0) is considered false, and nonzero values are true.
body1	The body of code to evaluate if the first test evaluates as true.
elseif	If *testExpression1* is false, evaluate *testExpression2*.
testExpression2	A second test to evaluate if the first test evaluates to false.
body2	The body of code to evaluate if the second test evaluates as true.
?else *bodyN?*	If all tests evaluate false, this body of code will be evaluated.

In the following example, note the placement of the curly braces ({}). The *Tcl Style Guide* describes the preferred format for `if`, `for`, `proc`, and `while` commands. It recommends that you place the left curly brace of the body of these commands on the line with the command and place the body on the next lines, indented two spaces. The final, right curly brace should go on a line by itself, indented even with the opening command. This makes the code less dense (and more easily read).

Putting the test and action on a single line is syntactically correct Tcl code but can cause maintenance problems later. You will need to make some multiline choice statements, and mixing multiline and single-line commands can make the action statements difficult to find. Also, what looks simple when you start writing some code may need to be expanded as you discover more about the problem you are solving. It is recommended practice to lay out your code to support adding new lines of code.

A Tcl command is normally terminated by a newline character. Thus, a left curly brace must be at the end of a line of code, not on a line by itself. Alternatively, you can write code with the new line escaped, and the opening curly brace on a new line, but this style makes code difficult to maintain.

Example 48
A Simple Test

```
set x 2
set y 3
if {$x < $y} {
  puts "x is less than y"
}
```

Script Output

```
x is less than y
```

The `switch` *Command*

The `switch` command allows a Tcl script to choose one of several patterns. The `switch` command is given a variable to test and several patterns. The first pattern that matches the test phrase will be evaluated, and all other sets of code will not be evaluated.

Syntax: `switch ?opt? str pat1 bod1 ?pat2 bod2 ...? ?default defaultBody?`

Evaluate 1 of N possible code bodies, depending on the value of a string.

?opt?	One of the following possible options.
	`-exact` Match a pattern string exactly to the test string, including a possible "-" character.
	`-glob` Match a pattern string to the test string using the `glob` string match rules. These are the default matching rules.
	`-regexp` Match a pattern string to the test string using the regular expression string match rules.
	`--` Absolutely the last option. The next string will be the string argument. This allows strings that start with a dash (`-`) to be used as arguments without being interpreted as options.
str	The string to match against patterns.
patN	A pattern to compare with the `string`.
bodN	A code body to evaluate if `patN` matches `string`.
default	A pattern that will match if no other patterns have matched.
defaultBody	The script to evaluate if no other patterns were matched.

The options `-exact`, `-glob`, and `-regexp` control how the `string` and `pattern` will be compared. By default, the `switch` command matches the patterns using the glob rules described previously with `string match`.

You can use regular expression match rules by including the `-regexp` flag. The regular expression rules are similar in that they allow you to define a pattern of characters in a string but are more complex and more powerful. The `regexp` command, which is used to evaluate regular expressions, is discussed in Chapter 5. The `switch` command can also be written with curly braces around the patterns and body.

Syntax:
```
switch ?option? string {
    pattern1 body1
    ?pattern2 body2?
    ?default defaultBody?
}
```

When the `switch` command is used without braces (as shown in the first `switch` statement below that follows), the `pattern` strings may be variables, allowing a script to modify the behavior of a `switch` command at runtime. When the braces are used (the second example following), the `pattern` strings must be hard-coded patterns.

Example 49

Script Example

```
set x 7
set y 7
# Using no braces substitution occurs before the switch
# command looks for matches.
# Thus a variable can be used as a match pattern:

switch $x \
  $y {puts "X=Y"} \
  {[0-9]} {puts "< 10"} \
  default {puts "> 10"}

# With braces, the $y is not substituted to 7, and switch looks
# for a match to the literal string $y

switch -glob $x {
  "1" {puts "one"}
  "2" {puts "two"}
  "3" {puts "three"}
  "$y" {puts "X=Y"}
  {[4-9]} {puts "greater than 3"}
  default {puts "Not a value between 1 and 9"}
}
```

Script Output

```
X=Y
greater than 3
```

If you wish to evaluate the same script when more than one pattern is matched, you can use a dash (`-`) in place of the body to cause the `switch` command to evaluate the next body, instead of the body associated with the current pattern. Part of a folk music quiz might resemble the following.

Example 50

Script Example

```
puts "Who recorded ‘Mr Tambourine Man’ "
gets stdin artist  ;# User types Bob Dylan
switch $artist {
  {Bob Dylan} -
  {Judy Collins} -
  {Glen Campbell} -
  {William Shatner} -
  {The Chipmunks} -
  {The Byrds} {
    puts "$artist recorded 'Mr Tambourine Man' "
  }
  default {
    puts "$artist probably recorded 'Mr Tambourine Man' "
  }
}
```

Script Output

```
Who recorded 'Mr Tambourine Man'
Bob Dylan
Bob Dylan recorded 'Mr Tambourine Man'
```

3.4.3 Looping

Tcl provides commands that allow a script to loop on a counter, loop on a condition, or loop through the items in a list. These three commands are as follows.

`for`	A numeric loop command
`while`	A conditional loop command
`foreach`	A list-oriented loop command

The `for` Command

The `for` command is the numeric loop command.

Syntax: `for` *`start test next body`*

Set initial conditions and loop until the *`test`* fails.

`start`	Tcl statements that define the start conditions for the loop.
`test`	A statement that tests an end condition. This statement must be in a format acceptable to `expr`.
`next`	A Tcl statement that will be evaluated after each pass through the loop. Normally this increments a counter.
`body`	The body of code to evaluate on each pass through the loop.

The `for` command is similar to the looping `for` in C, FORTRAN, BASIC, and others. The `for` command requires four arguments; the first (*`start`*) sets the initial conditions, the next (*`test`*) tests the condition, and the third (*`next`*) modifies the state of the test. The last argument (*`body`*) is the body of code to evaluate while the *`test`* returns true.

Example 51

Script Example

```
for {set i 0} {$i < 2} {incr i} {
  puts "I is: $i"
}
```

Script Output

```
I is: 0
I is: 1
```

The `while` Command

The `while` command is used to loop until a test condition becomes false.

Syntax: `while` *`test body`*

Loop until a condition becomes false.

`test`	A statement that tests an end condition. This statement must be in a format acceptable to `expr`.
`body`	The body of code to evaluate on each pass through the loop.

Example 52

While Loop Example

```
set x 0;
while {$x < 5} {
  set x [expr $x+$x+1]
  puts "X: $x"
}
```

Script Output

```
X: 1
X: 3
X: 7
```

The `foreach` Command

The `foreach` command is used to iterate through a list of items.

Syntax: `foreach` *`listVar list body`*

Evaluate *`body`* for each of the items in *`list`*.

`listVar`	This variable will be assigned the value of the list element currently being processed.
`list`	A list of data to step through.
`body`	The body of code to evaluate on each pass through the loop.

Example 53

Script Example

```
set total 0
foreach num {1 2 3 4 5} {
  set total [expr $total + $num]
}
puts "The total is: $total"
```

Script Output

```
The total is: 15
```

With Tcl release 7.5 (1996) and later, the `foreach` command was extended to handle multiple sets of list variables and list data.

Syntax: `foreach` *`valueList1 dataList1 ?valueList2 dataList2?...`* `{`
```
             body
         }
```

If the *`valueList`* contains more than one variable name, the Tcl interpreter will take enough values from the *`dataList`* to assign a value to each variable on each pass. If the *`dataList`* does not contain an even multiple of the number of *`valueList`* elements, the variables will be assigned an empty string.

Example 54

Script Example

```
foreach {pres date} { {George Washington} {1789-1797}
                {John Adams} {1797-1801}
                {Thomas Jefferson} {1801-1809}
                {James Madison} {1809-1817}
                {James Monroe} {1817-1825}
      } state {Virginia Massachusetts Virginia Virginia Virginia} {
  puts "$pres was from $state and served from $date"
}
```

Script Output

```
George Washington was from Virginia and served from 1789–1797
John Adams was from Massachusetts and served from 1797–1801
Thomas Jefferson was from Virginia and served from 1801–1809
James Madison was from Virginia and served from 1809–1817
James Monroe was from Virginia and served from 1817–1825
```

3.4.4 Exception Handling in Tcl

When the Tcl interpreter hits an exception condition the default action is to halt the execution of the script and display the data in the `errorInfo` global variable. The information in `errorInfo` will describe the command that failed and will include a stack dump for all the procedures that were in process when this failure occurred. The simplest method of modifying this behavior is to use the `catch` command to intercept the exception condition before the default error handler is invoked.

Syntax: `catch` *`script ?varName?`*

Catch an error condition and return the results rather than aborting the script.

`script`	The Tcl script to evaluate.
`varName`	Variable to receive the results of the script.

The `catch` command catches an error in a script and returns a success or failure code rather than aborting the program and displaying the error conditions. If the script runs without errors, `catch` returns 0. If there is an error, `catch` returns 1, and the `errorCode` and `errorInfo` variables are set to describe the error.

Sometimes a program should generate an exception. For instance, while checking the validity of user-provided data, you may want to abort processing if the data is obviously invalid. The Tcl command for generating an exception is `error`.

Syntax: `error` *`informationalString ?Info? ?Code?`*

`error`	Generate an error condition. If not caught, display the *`informationalString`* and stack trace and abort the script evaluation.
`informationalString`	Information about the error condition.
`Info`	A string to initialize the `errorInfo` string. Note that the Tcl interpreter may append more information about the error to this string.
`Code`	A machine-readable description of the error that occurred. This will be saved in the global `errorCode` variable.

The next example shows some ways of using the `catch` and `error` commands.

Example 55

Script Example

```
proc errorProc {first second} {
  global errorInfo
  # $fail will be non-zero if $first is non-numeric.
  set fail [catch {expr 5 * $first} result]
  # if $fail is set, generate an error
  if {$fail} {
    error "Bad first argument"
  }
  # This will fail if $second is non-numeric or 0
  set fail [catch {expr $first/$second} dummy]
  if {$fail} {
    error "Bad second argument" \
    "second argument fails math test\cback n\$errorInfo"
  }
  error "errorProc always fails" "evaluating error" \
    [list USER {123} {Non-Standard User-Defined Error}]
}
# Example Script
puts "call errorProc with a bad first argument"
set fail [catch {errorProc X 0} returnString]
```

```
if {$fail} {
  puts "Failed in errorProc"
  puts "Return string: $returnString"
  puts "Error Info: $errorInfo\n"
}
puts "call errorProc with a 0 second argument"
if {[catch {errorProc 1 0} returnString]} {
  puts "Failed in errorProc"
  puts "Return string: $returnString"
  puts "Error Info: $errorInfo\n"
}
puts "call errorProc with valid arguments"
set fail [catch {errorProc 1 1} returnString]
if {$fail} {
  if {[string first USER $errorCode] == 0} {
    puts "errorProc failed as expected"
    puts "returnString is: $returnString"
    puts "errorInfo: $errorInfo"
  } else {
    puts "errorProc failed for an unknown reason"
  }
}
```

Script Output

```
call errorProc with a bad first argument
Failed in errorProc
Return string: Bad first argument
Error Info: Bad first argument
  while executing
"error "Bad first argument""
  (procedure "errorProc" line 10)
  invoked from within
"errorProc X 0"
call errorProc with a 0 second argument
Failed in errorProc
Return string: Bad second argument
Error Info: second argument fails math test
divide by zero
  while executing
"expr \$first/\$second"
  (procedure "errorProc" line 15)
  invoked from within
"errorProc 1 0"
call errorProc with valid arguments
errorProc failed as expected
returnString is: errorProc always fails
errorInfo: evaluating error
  (procedure "errorProc" line 1)
```

```
    invoked from within}
  "errorProc 1 1"
```

Note the differences in the stack trace returned in `errorInfo` in the error returns. The first, generated with `error` *`message`*, includes the `error` command in the trace, whereas the second, generated with `error` *`message Info`*, does not.

If there is an *`Info`* argument to the `error` command, this string is used to initialize the `errorInfo` variable. If this variable is not present, Tcl uses the default initialization, which is a description of the command that generated the exception. In this case, that is the `error` command. If your application needs to include information that is already in the `errorInfo` variable, you can append that information by including `$errorInfo` in your message, as done with the second test.

The `errorInfo` variable contains what should be human-readable text to help a developer debug a program. The `errorCode` variable contains a machine-readable description to enable a script to handle exceptions intelligently. The `errorCode` data is a list in which the first field identifies the class of error (ARITH, CHILDKILLED, POSIX, and so on), and the other fields contain data related to this error. The gory details are in the on-line manual/help pages under `tclvars`.

If you are used to Java, you are already familiar with the concept of separating data returns from status returns. If your background is C/FORTRAN/BASIC type programming, you are probably more familiar with the C/FORTRAN paradigm of returning status as a function return, or using special values to distinguish valid data from error returns. For example, the C library routines return a valid pointer when successful, and a NULL pointer for failure.

If you want to use function return values to return status in Tcl, you can. Using the `error` command (particularly in low-level procedures that application programs will invoke) provides a better mechanism. The following are reasons for using `error` instead of status returns.

- An application programmer must check a procedure status return. It is easy to forget to check a status return and miss an exception. It takes extra code (the `catch` command) to ignore bad status generated by `error`.
- This makes the fast and dirty techniques for writing code (not checking for status, or not catching errors) the more robust technique. If a low-level procedure has a failure, the intermediate code must propagate the failure to the top level. Doing this with status returns requires special code to propagate the error, which means all functions must adhere to the error-handling policy.
- The `error` command automatically propagates the error. Procedures that use a function that may fail need not include exception propagation code. This moves the policy decisions for how to handle an exception to the application level, where it is more appropriate.

3.5 MODULARIZATION

Tcl has support for all modern software modularization techniques.

- Subroutines (with the `proc` command)
- Multiple source files (with the `source` command)
- Libraries (with the `package` command)

The `package` commands are discussed in detail in Chapter 8.

3.5.1 Procedures

The procedure is the most common technique for code modularization. Tcl procedures:

- Can be invoked recursively.
- Can be defined to accept specific arguments.
- Can be defined to accept arguments that have default values.
- Can be defined to accept a variable number of arguments.

The `proc` command defines a Tcl procedure.

Syntax: `proc` *`procName argList body`*

Defines a new procedure.

`procName`	The name of the procedure to define.
`argList`	The list of arguments for this procedure.
`body`	The body to evaluate when this procedure is invoked.

Note how the argument list and body are enclosed in curly braces in the following example. This is the normal way for defining a procedure, since you normally do not want any substitutions performed until the procedure body is evaluated. Procedures are discussed in depth in Chapter 7.

Example 56

Proc Example

```
# Define the classic recursive procedure to find the
# n'th position in a Fibonacci series.
proc fib {num} {
  if {$num <= 2} {return 1}
  return [expr [fib [expr $num -1]] + [fib [expr $num -2]] ]
}
for {set i 1} {$i < 6} {incr i} {
  puts "Fibonacci series element $i is: [fib $i]"
}
```

Script Output

```
fibonacci series element 1 is: 1
fibonacci series element 2 is: 1
fibonacci series element 3 is: 2
fibonacci series element 4 is: 3
fibonacci series element 5 is: 5
```

3.5.2 Loading Code from a Script File

Splitting functionality into separate files, so that each file contains closely related procedures, makes code easier to maintain. The `source` command loads a file into an existing Tcl script. It is similar to the `#include` in C, the `source` in C-shell programming, and the `require` in Perl. This command lets you build source code modules you can load into your scripts when you need particular functionality.

This allows you to modularize your programs. This is the simplest of the Tcl commands that implement libraries and modularization. The `package` command is discussed in Chapter 8.

Syntax: `source` *`fileName`*

Load a file into the current Tcl application and evaluate it.

`fileName`	The file to load.

Macintosh users have two options to the `source` command that are not available on other platforms.

Syntax: `source` *`-rsrc resourceName ?fileName?`*

Syntax: `source` *`-rsrcid resourceId ?fileName?`*

These options allow one script to source another script from a TEXT resource. The resource may be specified by `resourceName` or `resourceID`.

3.5.3 Examining the State of the Tcl Interpreter

Any Tcl script can query the Tcl interpreter about its current state. The interpreter can report whether a procedure or variable is defined, what a procedure body or argument list is, the current level in the procedure stack, and so on. The next examples will only use a few of the `info` subcommands. See the on-line documentation for details of the other subcommands.

Syntax: `info` *`subCommand arguments`*

Provide information about the interpreter state.

`subCommand` Defines the interaction. Interactions include:

`exists varName`	Returns `True` if a variable has been defined.
`proc globPattern`	Returns a list of procedure names that match the `glob` pattern.
`body procName`	Returns the body of a procedure.
`args` *`procName`*	Returns the names of the arguments for a procedure.
`nameofexecutable`	Returns the full path name of the binary file from which the application was invoked.

This example shows a procedure that counts the number of times values appear in a list and returns a list of values and the number of times they occur. It uses the `info exists` command to determine whether or not a value has been found (and counted) yet.

Example 57

Using* `info` *Commands

```
proc countListElements {lst} {
  # Step through each element in the list
  foreach l $lst {
      # If the index exists, increment the count
      if {[info exists counts($l)]} {
        incr counts($l)
      } else {
        # If the index did not exist, initialize it
        set counts($l) 1
      }
    }
    return [array get counts]
  }

  if {[info proc countListElements] eq "countListElements"} {
    set testList {a b a a c b a}
    puts "The countListElements procedure is defined."
    puts "The countListElements procedure returns"
    puts [countList $testList]
    puts "for the list {$testList}"
  }
```

Script Output

```
The countListElements procedure is defined.
The countListElements procedure returns
a 4 b 2 c 1
for the list {a b a a c b a}
```

3.6 BOTTOM LINE

This covers the basics of the Tcl language. The next chapter introduces the Tcl I/O calls, techniques for using these commands, and a few more commands.

- Tcl is a position-based language rather than a keyword-based language.
- A Tcl command consists of
  - A command name
  - Optional subcommand, flags, or arguments
  - A command terminator [either a newline or semicolon (;)]
- Words and symbols must be separated by at least one `whitespace` (space, tab, or escaped newline) character.
- Multiple words or variables can be grouped into a single argument with braces ({}) or quotes (" ").
- Substitution will be performed on strings grouped with quotes.
- Substitutions will not be performed on strings grouped with curly braces ({}).
- A Tcl command is evaluated in a single pass.
- The Tcl evaluation routine is called recursively to evaluate commands enclosed within square brackets.

- Some Tcl commands can accept flags to modify their behavior. A flag will always start with a hyphen. It may proceed or follow the arguments (depending on the command) and may require an argument itself.
- Values are assigned to a variable with the `set` command.
  Syntax: `set` *`varName value`*
- Math operations are performed with the `expr` and `incr` commands.
  Syntax: `expr` *`mathExpression`*
  Syntax: `incr` *`varName ?incrValue?`*
- The branch commands are `if` and `switch`.
  Syntax: `if` {*`test`*} {*`bodyTrue`*} *`?elseif`* {*`test2`*} {*`body2`*}*`? ?else`* {*`bodyFalse`*}*`?`*
  Syntax: `switch` *`?option? string pattern1 body1`*\
  `?pattern2 body2? ?default defaultBody?`
- The looping commands are `for`, `while`, and `foreach`.
  Syntax: `for` *`start test next body`*
  Syntax: `while` *`test body`*
  Syntax: `foreach` *`listVar list body`*
- The list operations include `list`, `split`, `llength`, `lindex`, and `lappend`.
  Syntax: `list` *`element1 ?element2? ... ?elementN?`*
  Syntax: `linsert` *`list position element1 ... ?elementN?`*
  Syntax: `lappend` *`listName ?element1? ... ?elementN?`*
  Syntax: `split` *`data ?splitChar?`*
  Syntax: `join` *`list ?joinString?`*
  Syntax: `llength` *`list`*
  Syntax: `lindex` *`list index`*
  Syntax: `lsearch` *`list pattern`*
  Syntax: `lreplace` *`list position1 position2 element1 ?... elementN?`*
- The string processing subcommands include `first`, `last`, `length`, `match`, `toupper`, `tolower`, and `range`.
  Syntax: `string first` *`substr string`*
  Syntax: `string last` *`substr string`*
  Syntax: `string length` *`string`*
  Syntax: `string match` *`pattern string`*
  Syntax: `string toupper` *`string`*
  Syntax: `string tolower` *`string`*
  Syntax: `string range` *`string first last`*
- Formatted strings can be generated with the `format` command.
  Syntax: `format` *`format ?data? ?data2? ...`*
- The `scan` command will perform simple string parsing.
  Syntax: `scan` *`textstring format ?varName1? ?varName2? ...`*
- The array processing subcommands include `array names`, `array set`, and `array get`.
  Syntax: `array names` *`arrayName ?pattern?`*
  Syntax: `array set` *`arrayName`* {*`index1 value1 ...`*}
  Syntax: `array get` *`arrayName`*

- Values can be converted between various ASCII and binary representations with `binary scan` and `binary format`.
- The `source` command loads and evaluates a script.
  Syntax: `source` *`fileName`*
- The `info` command returns information about the current state of the interpreter.
  Syntax: `info` *`proc`*
  Syntax: `info` *`args`*
  Syntax: `info` *`body`*
  Syntax: `info` *`exists`*
  Syntax: `info` *`nameofexecutable`*

3.7 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page, or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material, or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What will the following code fragments display?

  -
    ```
    set a 1
    puts "$a"
    ```

  -
    ```
    set a 1
    puts {$a}
    ```

  -
    ```
    set a 1
    puts [expr $a + 1]
    ```

  -
    ```
    set a b
    set $a 2
    puts "$b"
    ```

  -
    ```
    set a 1
    puts "\$$a"
    ```
- *101* What are the Tcl's three looping commands?
- *102* What conditional commands does Tcl support?
- *103* What command will define a Tcl procedure?

- *104* Can a Tcl procedure be invoked recursively?
- *105* What is the first word in a Tcl command line?
- *106* How are Tcl commands terminated?
- *107* Can you use binary data in Tcl?
- *108* How does a Tcl procedure return a failure status?
- *109* What commands can modify the content of a variable?
- *110* How could you change an integer to a floating-point value with the `append` command?
- *111* Write a pattern for `string match` to match:
  - Strings starting with the letter A.
  - Strings starting with the letter A followed by a number.
  - Strings starting with a lowercase letter followed by a number.
  - Strings in which the second character is a number.
  - Three character strings of uppercase letters.
  - A question.
- *112* What characters are returned by:
  - `string range "testing" 0 0`
  - `string range "testing" 0 1`
  - `string range "testing" 0 99`
  - `string range "testing" 0 end`
  - `string range "testing" 99 end`
  - `string range "testing" end end`
- *113* Write a format definition that will:
  - Use 20 spaces to display a string, and left justify the string.
  - Use 20 spaces to display a string, and right justify the string.
  - Display a floating point number less than 100 with 2 digits to the right of the decimal point.
  - Display a floating point number in scientific notation.
  - Convert an integer to an ASCII character (i.e., convert 48 to ”0”, 49 to ”1”, and so on).
- *114* What is the second list element in the following lists?
  - {one two three four}
  - {one {two three} four}
  - { {} one two three}
  - { {one two} {three four} }
- *115* Which array command will return a list of the indices in an associative array?
- *116* Which Tcl command could be used to assign a value to a single element in an associative array?
- *117* Which Tcl command could be used to assign values to multiple elements in an associative array?

- *118* What `binary scan` format definition would read data that was written as this C structure?: struct { int i[4]; char c[25]; float f; }
- *119* If `x` and `y` are two Tcl variables containing the length of the opposite and adjacent sides of a triangle, write the `expr` command that would calculate the hypotenuse of this angle.
- *120* Write a procedure that uses info commands to report all the procedures and arguments that exist in an interpreter.
- *200* The classic recursive function is a Fibonacci series, in which each element is the sum of the two preceding elements, as in the following.
  ```
  1 1 2 3 5 8 13 21 ..
  ```
  Write a Tcl proc that will accept a single integer, and will generate that many elements of a Fibonacci series.
- *201* Write a procedure that will accept a string of text, and will generate a histogram of how many times each unique word is used in that text.
- *202* Write a procedure that will accept a set of comma-delimited lines, and will generate a formatted table from that data.
- *203* Write a procedure that will check to see whether a string is a palindrome (if it reads the same backward and forward). Examples of palindromes include the words *noon* and *radar*, and the classic sentence *Able was I ere I saw Elba*.
- *300* The bubble sort works by stepping through a list and comparing two adjacent members and swapping them if they are not in ascending order. The list is scanned repeatedly until there are no more elements in the wrong position.
  a. Write a recursive procedure to perform a bubble sort on a list of data.
  b. Write a loop-based procedure to perform a bubble sort on a list of data.
- *301* Tcl has an `lsort` command that will sort a list. Use the `lsort` command to check the results of the bubble sort routines constructed in the previous exercise.
- *302* A trivial encryption technique is to group characters in sets of four, convert that to an integer, and print out the integers. Write a pair of Tcl procedures that will use the `binary` command to convert a plaintext message to a list of integers, and convert a list of integers into a readable string.