CHAPTER 1

Tcl Fundamentals

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

This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.

http://www.beedub.com/book/

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

This chapter takes you through the basics of the Tcl language syntax. Even if you are an expert programmer, it is worth taking the time to read these few pages to make sure you understand the fundamentals of Tcl. The basic mechanisms are all related to strings and string substitutions, so it is fairly easy to visualize what is going on in the interpreter. The model is a little different from some other programming languages with which you may already be familiar, so it is worth making sure you understand the basic concepts.

Tcl Commands

Tcl stands for Tool Command Language. A command does something for you, like output a string, compute a math expression, or display a widget on the screen. Tcl casts everything into the mold of a command, even programming constructs

like variable assignment and procedure definition. Tcl adds a tiny amount of syntax needed to properly invoke commands, and then it leaves all the hard work up to the command implementation.

The basic syntax for a Tcl command is:

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

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

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

This model is described in detail in this Chapter.

Hello, World!

Example 1–1 The "Hello, World!" example.

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

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

- Arguments are interpreted by the command. In the example, `stdout` is used to identify the standard output stream. The use of `stdout` as a name is a convention employed by `puts` and the other I/O commands. Also, `stderr` is used to identify the standard error output, and `stdin` is used to identify the standard input. Chapter 9 describes how to open other files for I/O.
- Curly braces are used to group words together into a single argument. The `puts` command receives `Hello, World!` as its second argument.

The braces are not part of the value.

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

Variables

The `set` command is used to assign a value to a variable. It takes two arguments: The first is the name of the variable, and the second is the value. Variable names can be any length, and case *is* significant. In fact, you can use any character in a variable name.

It is not necessary to declare Tcl variables before you use them.

The interpreter will create the variable when it is first assigned a value. The value of a variable is obtained later with the dollar-sign syntax, illustrated in Example 1–2:

Example 1–2 Tcl variables.

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

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

```
set b 5
```

The actual implementation of substitution is more efficient, which is important when the value is large.

Command Substitution

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

Example 1–3 Command substitution.

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

In Example 1–3, the nested command is:

```
string length foobar
```

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

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

```
set len 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered, the command it delimits is evaluated. This results in a sensible ordering in which nested commands are evaluated first so that their result can be used in arguments to the outer command.

Math Expressions

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

Example 1–4 Simple arithmetic.

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

The math syntax supported by `expr` is the same as the C expression syntax. The `expr` command deals with integer, floating point, and boolean values. Logical operations return either 0 (false) or 1 (true). Integer values are promoted to floating point values as needed. Octal values are indicated by a leading zero (e.g., `033` is `27` decimal). Hexadecimal values are indicated by a leading `0x`. Scientific notation for floating point numbers is supported. A summary of the operator precedence is given on page 20.

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

Example 1–5 Nested commands.

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

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

Example 1–6 Built-in math functions.

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

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

Example 1–7 Grouping expressions with braces.

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

Backslash Substitution

The final type of substitution done by the Tcl interpreter is *backslash substitution*. This is used to quote characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. As a rule, however, if you find yourself using lots of backslashes, there is probably a simpler way to achieve the effect you are striving for. In particular, the list command described on page 61 will do quoting for you automatically. In Example 1–8 backslash is used to get a literal $:

Example 1–8 Quoting special characters with backslash.

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

Only a single round of interpretation is done.

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

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

```
set escape \u001b
set escape \0x1b
set escape \033
```

The value of variable escape is the ASCII ESC character, which has character code 27. The table on page 20 summarizes backslash substitutions.

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

Example 1–9 Continuing long lines with backslashes.

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

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

Grouping with Braces and Double Quotes

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

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

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

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

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

Another common use of quotes is with the `format` command. This is similar to the C `printf` function. The first argument to `format` is a format specifier that often includes special characters like newlines, tabs, and spaces. The easiest way to specify these characters is with backslash sequences (e.g., `\n` for newline and `\t` for tab). The backslashes must be substituted before the `format` command is

called, so you need to use quotes to group the format specifier.

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

Here `format` is used to align a name and a value with a tab. The `%s` and `%5.3f` indicate how the remaining arguments to `format` are to be formatted. Note that the trailing `\n` usually found in a C `printf` call is not needed because `puts` provides one for us. For more information about the `format` command, see page 52.

Square Brackets Do Not Group

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

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

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

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

The following is a redundant use of double quotes:

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

Grouping before Substitution

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

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

Example 1–11 Embedded command and variable substitution.

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

In Example 1–11, the second argument to `puts` is:

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

The white space inside the nested command is ignored for the purposes of grouping the argument. By the time Tcl encounters the left bracket, it has already done some variable substitutions to obtain:

```
7+9=
```

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

```
7+9=16
```

Grouping before substitution.

The point of this example is that the grouping decision about `puts`'s second argument is made before the command substitution is done. Even if the result of the nested command contained spaces or other special characters, they would be ignored for the purposes of grouping the arguments to the outer command. Grouping and variable substitution interact the same as grouping and command substitution. Spaces or special characters in variable values do not affect grouping decisions because these decisions are made before the variable values are substituted.

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

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

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

Grouping Math Expressions with Braces

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

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

More Substitution Examples

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

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

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

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

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

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

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

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

Procedures

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

```
proc name arglist body
```

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

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

Example 1–12 Defining a procedure.

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

The Diag procedure defined in the example computes the length of the diagonal side of a right triangle given the lengths of the other two sides. The sqrt function is one of many math functions supported by the expr command. The variable c is local to the procedure; it is defined only during execution of Diag. Variable scope is discussed further in Chapter 7. It is not really necessary to use the variable c in this example. The procedure can also be written as:

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

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

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

Note the stylized use of curly braces in the example. The curly brace at the end of the first line starts the third argument to proc, which is the command body. In this case, the Tcl interpreter sees the opening left brace, causing it to ignore newline characters and scan the text until a matching right brace is found. *Double quotes have the same property.* They group characters, including newlines, until another double quote is found. The result of the grouping is that

the third argument to proc is a sequence of commands. When they are evaluated later, the embedded newlines will terminate each command.

The other crucial effect of the curly braces around the procedure body is to delay any substitutions in the body until the time the procedure is called. For example, the variables a, b, and c are not defined until the procedure is called, so we do not want to do variable substitution at the time Diag is defined.

The proc command supports additional features such as having variable numbers of arguments and default values for arguments. These are described in detail in Chapter 7.

A Factorial Example

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

Example 1–13 A while loop to compute factorial.

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

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

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

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

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

Always group expressions and command bodies with curly braces.

Curly braces around the boolean expression are crucial because they delay variable substitution until the `while` command implementation tests the expression. The following example is an infinite loop:

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

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

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

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

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

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

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

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

Example 1–14 A recursive definition of factorial.

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

More about Variables

The `set` command will return the value of a variable if it is only passed a single argument. It treats that argument as a variable name and returns the current value of the variable. The dollar-sign syntax used to get the value of a variable is really just an easy way to use the `set` command. Example 1–15 shows a trick you can play by putting the name of one variable into another variable:

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

Example 1–15 Using `set` to return a variable value.

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

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

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

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

Funny Variable Names

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

If the variable reference is not delimited by punctuation or white space, then you can use curly braces to explicitly delimit the variable name (e.g., `${x}`). You can also use this to reference variables with funny characters in their name, although you probably do not want variables named like that. If you find yourself using funny variable names, or computing the names of variables, then you may want to use the `upvar` command.

Example 1–16 Embedded variable references.

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

The `unset` Command

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

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

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

Using `info` to Find Out about Variables

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

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

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

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

More about Math Expressions

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

Example 1–18 Controlling precision with `tcl_precision`.

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

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

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

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

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

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

Grouping expressions can make them run more efficiently.

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

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

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

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

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

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

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

Comments

Tcl uses the pound character, `#`, for comments. Unlike in many other languages, the # must occur at the beginning of a command. A # that occurs elsewhere is not treated specially. An easy trick to append a comment to the end of a command is

to precede the # with a semicolon to terminate the previous command:

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

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

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

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

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

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

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

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

Substitution and Grouping Summary

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

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

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

- Grouping decisions are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A dollar sign, `$`, causes variable substitution. Variable names can be any length, and case is significant. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the `${`*`varname`*`}` syntax.
- Square brackets, `[ ]`, cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.
- The backslash character, `\`, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character or group of characters are replaced with a new character.
- Substitutions can occur anywhere unless prevented by curly brace grouping. Part of a group can be a constant string, and other parts of it can be the result of substitutions. Even the command name can be affected by substitutions.
- A single round of substitutions is performed before command invocation. The result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that contains special characters such as spaces, dollar signs, square brackets, or braces. Because only a single round of substitution is done, you do not have to worry about special characters in values causing extra substitutions.

Fine Points

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

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

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

```
set silly a"b
```

- When double quotes are used for grouping, the special effect of curly braces is turned off. Substitutions occur everywhere inside a group formed with

double quotes. In the next command, the variables are still substituted:

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

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

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

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

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

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

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

- During command substitution, newlines and semicolons *are* significant as command terminators. If you have a long command that is nested in square brackets, put a backslash before the newline if you want to continue the command on another line. This was illustrated in Example 1–9 on page 8.
- A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following sets `x` to the single character `$`.

```
set x $
```

Reference

Backslash Sequences

Table 1–1 Backslash sequences.

`\a`	Bell. (`0x7`)
`\b`	Backspace. (`0x8`)
`\f`	Form feed. (`0xc`)
`\n`	Newline. (`0xa`)
`\r`	Carriage return. (`0xd`)
`\t`	Tab. (`0x9`)
`\v`	Vertical tab. (`0xb`)
`\<newline>`	Replace the newline and the leading white space on the next line with a space.
`\\`	Backslash. (‘\’)
`\`*ooo*	Octal specification of character code. 1, 2, or 3 digits.
`\x`*hh*	Hexadecimal specification of character code. 1 or 2 digits.
`\u`*hhhh*	Hexadecimal specification of a 16-bit Unicode character value. 4 hex digits.
`\`*c*	Replaced with literal *c* if *c* is not one of the cases listed above. In particular, `\$`, `\"`, `\{`, `\}`, `\]`, and `\[` are used to obtain these characters.

Arithmetic Operators

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

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

Built-in Math Functions

Table 1–3 Built-in math functions.

`acos(x)`	Arccosine of *x*.
`asin(x)`	Arcsine of *x*.
`atan(x)`	Arctangent of *x*.
`atan2(y,x)`	Rectangular `(x,y)` to polar `(r,th)`. `atan2` gives *th*.
`ceil(x)`	Least integral value greater than or equal to *x*.
`cos(x)`	Cosine of *x*.
`cosh(x)`	Hyperbolic cosine of *x*.
`exp(x)`	Exponential, e^x.
`floor(x)`	Greatest integral value less than or equal to *x*.
`fmod(x,y)`	Floating point remainder of *x/y*.
`hypot(x,y)`	Returns `sqrt(x*x + y*y)`. *r* part of polar coordinates.
`log(x)`	Natural log of *x*.
`log10(x)`	Log base 10 of *x*.
`pow(x,y)`	*x* to the *y* power, x^y.
`sin(x)`	Sine of *x*.
`sinh(x)`	Hyperbolic sine of *x*.
`sqrt(x)`	Square root of *x*.
`tan(x)`	Tangent of *x*.
`tanh(x)`	Hyperbolic tangent of *x*.
`abs(x)`	Absolute value of *x*.
`double(x)`	Promote *x* to floating point.
`int(x)`	Truncate *x* to an integer.
`round(x)`	Round *x* to an integer.
`rand()`	Return a random floating point value between 0.0 and 1.0.
`srand(x)`	Set the seed for the random number generator to the integer *x*.

Core Tcl Commands

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

Table 1–4 Built-in Tcl commands.

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

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

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

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

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