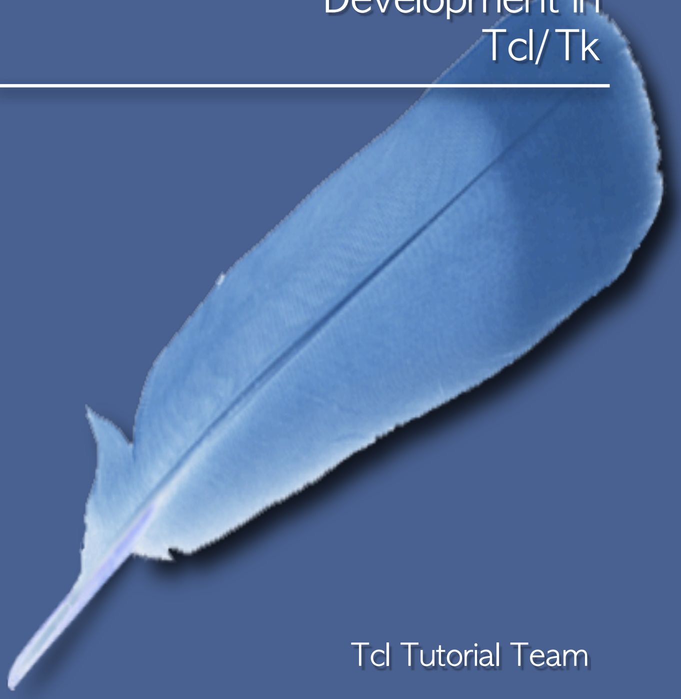
Modern Application Development in Tcl/Tk



Tcl Tutorial Team

Modern Application Development in Tcl/Tk

Tcl Tutorial Team

July 3, 2017

Contents

Chapter 1

Getting started

1.1 Introduction

Welcome to the Tcl tutorial. We wrote it with the goal of helping you to learn Tcl. It is aimed at those who have some knowledge of programming, although you certainly don't have to be an expert. The tutorial is intended as a companion to the Tcl manual pages which provide a reference for all Tcl commands.

It is divided into brief sections covering different aspects of the language. Depending on what system you are on, you can always look up the reference documentation for commands that you are curious about. On Unix for example, `man while` would bring up the man page for the `while` command.

Each section is accompanied by relevant examples showing you how to put to use the material covered.

Additional Resources

The Tcl community is an exceedingly friendly one. It's polite to try and figure things out yourself, but if you're struggling, we're more than willing to help. Here are some good places to get help:

- The comp.lang.tcl newsgroup. Accessible via a newsreader, or Google Groups.

- The *Wiki* (`http://wiki.tcl.tk`) has a great deal of useful code, examples and discussions of the finer points of Tcl usage.

- If you need help right away, there is often someone on the #tcl channel on irc.freenode.net who can help you out, but please don't be impatient if no one can help you instantly - if you need that level of support, consider hiring a consultant.

- There are several recommended books for those who wish to gain more in-depth knowledge of Tcl. *Clif Flynt* (`http://noucorp.com/Clif.html`), the original author of this tutorial is also the author of *Tcl/Tk: A Developer's Guide* (`http://www.msen.com/~clif/DevGuide.html`). Other popular books: *Practical Programming in Tcl and Tk* (`http://www.beedub.com/book/`).

Credits

Thanks first and foremost to Clif Flynt for making his material available under a BSD license. The following people also contributed:

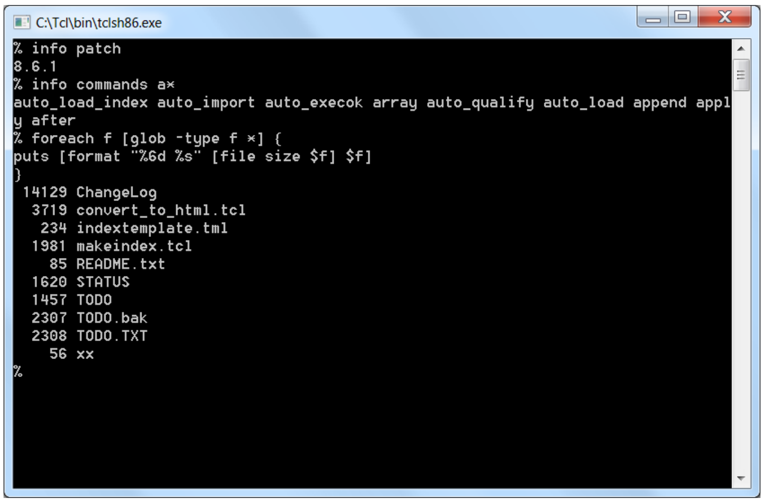
- *Neil Madden* (`http://www.cs.nott.ac.uk/~nem`)
- Arjen Markus
- *David N. Welton* (`http://www.dedasys.com/davidw`)

Of course, we also welcome comments and suggestions about how it could be improved - or if it's great the way it is, we don't mind a bit of thanks, either!

1.2 Running Tcl

When you have installed Tcl, the program you will then call to utilize it is `tclsh`. For instance, if you write some code to a file "hello.tcl", and you want to execute it, you would do it like so: `tclsh hello.tcl`. Depending on the version of Tcl installed, and the operating system distribution you use, the `tclsh` program may be a link to the real executable, which may be named `tclsh8.6` or `tclsh86.exe` on Microsoft Windows.

The `tclsh` program runs Tcl programs but it also allows interactive use: You can either start it with a script file on the command line, in which case it runs the script to completion and then exits, or you can start it without any arguments, in which case you get an interactive prompt, usually a `\%` symbol where you type in commands. Tcl will then execute and display the result, or any error messages that result. To exit the interpreter, type `exit` and press Return. For example:



```
% info patch
8.6.1
% info commands a*
auto_load_index auto_import auto_execok array auto_qualify auto_load append appl
y after
% foreach f [glob -type f *] {
puts [format "%6d %s" [file size $f] $f]
}
 14129 ChangeLog
  3719 convert_to_html.tcl
   234 indextemplate.tml
  1981 makeindex.tcl
    85 README.txt
  1620 STATUS
  1457 TODO
  2307 TODO.bak
  2308 TODO.TXT
    56 xx
%
```

Playing around with the interactive interpreter is a great way to learn how to use Tcl. Most Tcl commands will produce a helpful error message explaining how they are used if you just type in the command with no arguments. You can get a list of all the commands that your interpreter knows about by typing `info commands`.

In the various lessons we will use the conventions that *interactive sessions* are characterised by the `\%` prompt. The commands are preceded by the percent sign and the result is shown directly below, as it would appear if you type them in in `tclsh`.

The `tclsh` executable is just one way of starting a Tcl interpreter. Another common executable, which may be installed on your system, is the `wish`, or WIndowing SHell. This is a version of Tcl that automatically loads the Tk extension for building graphical user interfaces (GUIs). This tutorial does not cover Tk, and so we will not use the `wish` interpreter here. Other options are also available, providing more functional environments for developing and debugging code than that provided by the standard `tclsh`. One very popular choice is the *TkCon* (`http://tkcon.sourceforge.net/`) enhanced interpreter, written by Jeff Hobbs. The Eclipse IDE offers good Tcl support, in the form of the *DLTK* (`http://www.eclipse.org/dltk`) extension, and the Tcl'ers Wiki offers *a list of IDEs with Tcl support* (`http://wiki.tcl.tk/998`) and a comprehensive *catalogue of Tcl source code editors* (`http://wiki.tcl.tk/1184`). Don't panic, though! If you don't know how to use a sophisticated development environment, it is still very easy to write Tcl code by hand in a simple text editor (such as Notepad).

1.3 Simple Text Output

The traditional starting place for a tutorial is the classic "Hello, World" program. Once you can print out a string, you're well on your way to using Tcl for fun and profit!

The command to output a string in Tcl is the `puts` command.

A single word after the `puts` command will be printed to the standard output device. Normally the next text will be printed on the next line. The two commands

```
puts Hello,
puts World
```

produces:

```
Hello,
World
```

To have them printed on the same line, use the `-nonewline` option:

```
puts -nonewline Hello,
puts World
```

But as the string we want to print has more than one word, it will be easier to enclose the string in double quotes or braces (). A set of words enclosed in quotes or braces is treated as a single unit, while words separated by whitespace are treated as multiple arguments to the command:

```
puts "Hello, World"
```

Quotes and braces can both be used to group several words into a single unit. However, they actually behave differently. In the next lesson you'll start to learn some of the differences between their behaviors. *Note* that in Tcl, single quotes are not significant, as they are in other programming languages such as C, Perl and Python.

Many commands in Tcl (including `puts`) can accept multiple arguments. If a string is not enclosed in quotes or braces, the Tcl interpreter will consider each word in the string as a separate argument, and pass each individually to the command.

A full command in Tcl is the command name - the first word, in the example above `puts` - followed by a list of words terminated by a newline or semicolon. Tcl comments are indicated by a \# at a position where Tcl expects a new command (i.e., following a newline or semicolon), and continue until the end of the line.

Example

```
puts "Hello, World - In quotes" ;# This is a comment after the command.
# This is a comment at beginning of a line
puts {Hello, World - In Braces}

puts "This is line 1"; puts "this is line 2"

puts "Hello, World; - With a semicolon inside the quotes"

# Words don't need to be quoted unless they contain white space:
puts HelloWorld
```

Result:

```
Hello, World - In quotes
Hello, World - In Braces
This is line 1
this is line 2
Hello, World; - With a semicolon inside the quotes
HelloWorld
```

As stated, comments may appear wherever a new command can be expected. The following is a syntactic error - there should be a semicolon before the hash-sign:

```
puts {Bad comment syntax example} # *Error* - there is no semicolon!
```

Result:

```
wrong # args: should be "puts ?-nonewline? ?channelId? string"
    while executing
"puts {Bad comment syntax example} # *Error* - there is no semicolon!"
    (file "example.tcl" line 1)
```

This also illustrates the habit of Tcl to inform you about (run-time) errors. *This will be explained later.* (Section 7.5)

1.4 Assigning values to variables

In Tcl, everything may be represented as a string, although internally it may be represented as a list, integer, double, or some other type, in order to make the language fast.

The assignment command in Tcl is `set`.

When `set` is called with two arguments, as in:

```
 set fruit Cauliflower
```

it assigns the value `Cauliflower` (the second argument) to the variable `fruit` (the first argument). `set` always returns the contents of the variable named in the first argument. Thus, when `set` is called with two arguments, it assigns the second argument to the variable named in the first argument and then returns the second argument. In the above example, for instance, it would return "Cauliflower", without the quotes.

The first argument to a `set` command can be either a single word, like `fruit` or `pi` , or it can be a member of an array, like `course(first)`, an element called "first" in the array "course". Technically speaking, arrays in Tcl are *associative* arrays. Arrays will be discussed in greater detail later.

`set` can also be invoked with only one argument. When called with just one argument, it will return the contents of that argument.

Here's a summary of the `set` command.

`set varName ?value?`
If `value` is specified, then the contents of the variable `varName` are set equal to `value`.

- If `varName` consists only of alphanumeric characters, and no parentheses, it is a scalar variable.
- If `varName` has the form `varName(index)` , it is a member of an associative array.

If you look at the example code, you'll notice that in the `set` command the first argument is typed with only its name, but in the `puts` statement the argument is preceded with a `\$`.

The dollar sign tells Tcl to use the value of the variable - in this case `X` or `Y`.

Tcl passes data to subroutines either by name or by value. Commands that don't change the contents of a variable usually have their arguments passed by value. Commands that **do** change the value of the data must have the data passed by name.

Example

```
set X "This is a string"

set Y 1.24

puts $X
puts $Y
puts "..............................."

set label "The value in Y is: "
puts "$label $Y"
```

Result:

```
This is a string
1.24
...............................
The value in Y is: 1.24
```

1.5 Evaluation and Substitutions 1: Grouping arguments with ""

This lesson is the first of three which discuss the way Tcl handles substitution during command evaluation.

In Tcl, the evaluation of a command is done in 2 phases. The first phase is a single pass of substitutions. The second phase is the evaluation of the resulting command. Note that only *one* pass of substitutions is made. Thus in the command

```
    puts $varName
```

the contents of the variable `varName` are substituted for `\$varName`, and then the command is executed. Assuming we have set `varName` to `"Hello World"`, the sequence would look like this:

```
    puts $varName --> puts "Hello World"
```

, which is then executed and prints out `Hello World`.

During the substitution phase, several types of substitutions occur.

A command within square brackets ([]) is replaced with the result of the execution of that command. (This will be explained more fully in the lesson *Results of a Command - Math 101.* (Section 1.8))

Words within double quotes or braces are grouped into a single argument. However, double quotes and braces cause different behavior during the substitution phase. In this lesson, we will concentrate on the behavior of double quotes during the substitution phase.

Double quotes

Grouping words within double quotes allows substitutions to occur within the quotations - or, in fancier terms, "interpolation". The substituted group is then evaluated as a single argument. Thus, in the command:

```
    puts "The current stock value is $varName"
```

the current contents of varName are substituted for $varName, and then the entire string is printed to the output device, just like the example above.

Backslashes

In general, the backslash (
) disables substitution for the single character immediately following the backslash. Any character immediately following the backslash will stand without substitution.

However, there are specific "Backslash Sequence" strings which are replaced by specific values during the substitution phase. The following backslash strings will be substituted as shown below.

String	Output	Hex value
`\\a`	Audible Bell	0x07
`\\b`	Backspace	0x08
`\\f`	Form Feed (clear screen)	0x0c
`\\n`	New Line	0x0a
`\\r`	Carriage Return	0x0d
`\\t`	Tab	0x09
`\\v`	Vertical Tab	0x0b
`\\0dd`	Octal Value	d is a digit from 0-7
`\\uHHHH`	H is a hex digit 0-9,A-F,a-f.	
	This represents a 16-bit Unicode character.	
`\\xHH....`	Hexadecimal Value	H is a hex digit 0-9,A-F,a-f.

Note that the
x substitution "keeps going" as long as it has hexadecimal digits, and only uses the last two, meaning that
xaa and
xaaaa are equal, and that
xaaAnd anyway will "eat" the A of "And". Using the
u notation is probably a better idea.

The final exception is the backslash at the end of a line of text. This causes the interpreter to ignore the newline, and treat the text as a single line of text. The interpreter will insert a blank space at the location of the ending backslash.

Example

```
set Z Albany
set Z_LABEL "The Capitol of New York is: "

puts "$Z_LABEL $Z" ;# Prints the value of Z
puts "$Z_LABEL \$Z" ;# Prints a literal $Z instead of the value of Z

puts "\nBen Franklin is on the \$100.00 bill"

set a 100.00
puts "Washington is not on the $a bill" ;# This is not what you want
puts "Lincoln is not on the $$a bill" ;# This is OK
puts "Hamilton is not on the \$a bill" ;# This is not what you want
puts "Ben Franklin is on the \$$a bill" ;# But, this is OK
puts "\n................ examples of escape strings"
puts "Tab\tTab\tTab"
puts "This string prints out \non two lines"
puts "This string comes out\
on a single line"
```

Result:

```
The Capitol of New York is: Albany
The Capitol of New York is: $Z

Ben Franklin is on the $100.00 bill
Washington is not on the 100.00 bill
Lincoln is not on the $100.00 bill
Hamilton is not on the $a bill
Ben Franklin is on the $100.00 bill

................ examples of escape strings
Tab Tab Tab
This string prints out
on two lines
This string comes out on a single line
```

1.6 Evaluation and Substitutions 2: Grouping arguments with

During the substitution phase of command evaluation, the two grouping operators, the brace () and the double quote ("), are treated differently by the Tcl interpreter.

In the last lesson you saw that grouping words with double quotes allows substitutions to occur within the double quotes. By contrast, grouping words within double braces **disables** substitution within the braces. Characters within braces are passed to a command exactly as written. The only "Backslash Sequence" that is processed within braces is the backslash at the end of a line. This is still a line continuation character.

Note that braces have this effect only when they are used for grouping (i.e. at the beginning and end of a sequence of words). If a string is already grouped, either with quotes or braces, and braces occur in the middle of the grouped string (i.e. "foobar"), then the braces are treated as regular characters with no special meaning. If the string is grouped with quotes, substitutions will occur within the quoted string, even between the braces.

Example

```
set Z Albany
set Z_LABEL "The Capitol of New York is: "

puts "\n................ examples of differences between \" and \{"
puts "$Z_LABEL $Z"
puts {$Z_LABEL $Z}

puts "\n....... examples of differences in nesting \{ and \" "
puts "$Z_LABEL {$Z}"
puts {Who said, "What this country needs is a good $0.05 cigar!"?}

puts "\n................ examples of escape strings"
puts {There are no substitutions done within braces \n \r \x0a \f \v}
puts {But, the escaped newline at the end of a\
string is still evaluated as a space}
```

Result:

```
................ examples of differences between " and {
The Capitol of New York is: Albany
$Z_LABEL $Z

....... examples of differences in nesting { and "
The Capitol of New York is: {Albany}
Who said, "What this country needs is a good $0.05 cigar!"?

................ examples of escape strings
There are no substitutions done within braces \n \r \x0a \f \v
But, the escaped newline at the end of a string is still evaluated as a space
```

1.7 Evaluation and Substitutions 3: Grouping arguments with []

You obtain the results of a command by placing the command in square brackets ([]). This is the functional equivalent of the back single quote `() in sh programming, or using the return valu`

As the Tcl interpreter reads in a line it replaces all the $variables with their values. If a portion of the string is grouped with square brackets, then the string within the square brackets is evaluated as a command by the interpreter, and the result of the command replaces the square bracketed string.

```
   puts [readsensor [selectsensor]]
```

- The parser scans the entire command, and sees that there is a command substitution to perform: `readsensor [selectsensor]` , which is sent to the interpreter for evaluation.
- The parser once again finds a command to be evaluated and substituted, `selectsensor`
- The fictitious `selectsensor` command is evaluated, and it presumably returns a sensor to read.
- At this point, readsensor has a sensor to read, and the readsensor command is evaluated.

- Finally, the value of readsensor is passed on back to the `puts` command, which prints the output to the screen.

The exceptions to this rule are as follows:

- A square bracket that is escaped with a
  is considered as a literal square bracket.
- A square bracket within braces is not modified during the substitution phase.

Example

```
set x abc
puts "A simple substitution: $x\n"

set y [set x "def"]
puts "Remember that set returns the new value of the variable: X: $x Y: $y\n"

set z {[set x "This is a string within quotes within braces"]}
puts "Note the curly braces: $z\n"

set a "[set x {This is a string within braces within quotes}]"
puts "See how the set is executed: $a"
puts "\$x is: $x\n"

set b "\[set y {This is a string within braces within quotes}]"
# Note the \ escapes the bracket, and must be doubled to be a
# literal character in double quotes
puts "Note the \\ escapes the bracket:\n \$b is: $b"
puts "\$y is: $y"
```

Result:

```
A simple substitution: abc

Remember that set returns the new value of the variable: X: def Y: def

Note the curly braces: [set x "This is a string within quotes within braces"]

See how the set is executed: This is a string within braces within quotes
$x is: This is a string within braces within quotes

Note the \ escapes the bracket:
 $b is: [set y {This is a string within braces within quotes}]
$y is: def
```

1.8 Results of a command - Math 101

The Tcl command for doing mathematical calculations is `expr`. The following discussion of the `expr` command is extracted and adapted from the `expr` man page. Many commands use `expr` behind the scenes in order to evaluate test expressions, such as `if`, `while` and `for` loops, discussed in later sections. All of the advice given here for `expr` also holds for these other commands.

`expr` takes all of its arguments (”2 + 2” for example) and evaluates the result as a Tcl ”expression” (rather than a normal command), and returns the value. The operators permitted in Tcl expressions include all the standard math functions, logical operators, bitwise operators, as well as math functions like `rand()`, `sqrt()`, `cosh()` and so on. Expressions almost always yield numeric results (integer or floating-point values).

Performance tip: enclosing the arguments to `expr` in curly braces will result in faster code. So do `expr {\$i * 10}` instead of simply `expr \$i * 10`. It is also safer, as illustrated at the end of this lesson.

Operands

A Tcl expression consists of a combination of operands, operators, and parentheses. *White space* may be used between operands, operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is 0), or in hexadecimal (if the first two characters of the operand are 0x).

Note that the octal and hexadecimal conversion takes place differently in the `expr` command than in the Tcl substitution phase. In the substitution phase, a `\\x32` would be converted to an ascii ”2”, while `expr` would convert `0x32` to a decimal 50.

If an operand does not have one of the integer formats given above, then it is treated as a floating-point number, if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler. For example, all of the following are valid floating-point numbers:

```
2.1
3.
6E4
7.91e+16
.000001
```

If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it) but literal strings must be enclosed in double quotes.

Consider the following example: the variable `number` has a value `2,1`, which is not interpreted as a valid numerical value. So instead it is regarding as a string. Then any operation is done with strings in mind.

```
% set number 2,1
% expr {$number > 2} ;# True, because the string "2,1" alphabetically comes after "2"
1
% expr {$number > 2.0} ;# False, because the string "2,1" alphabetically comes before "2.0"
0
```

It is possible to deal with numbers in that form, but you will have to convert these ”strings” to numbers in the standard form first.

Operands may be specified in any of the following ways:

- As a numeric value, either integer or floating-point.
- As strings enclosed in double quotes.
- As a boolean (logical) value, `1, 0, true, false`
- As a Tcl variable, using standard $ notation. The variable's value will be used as the operand.

Operators

Some operators work on numbers in general, some work on integer numbers only and some work on strings only. Each category is listed below

Operations on numbers

The operators that work on numbers of any kind are listed below, grouped in decreasing order of precedence:

`- +`
Unary minus, unary plus

`**`
Exponentiation (works on both floating-point numbers and integers)

`* /`
Multiply, divide. For integers, see below

`+ -`
Add and subtract.

`== != < > <= >= <`
Relational operators: is equal, not equal, less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to numeric operands as well as strings, in which case string comparison is used.

```
% set x 1
% expr { $x>0? ($x+1) : ($x-1) }
2
```

Operations on integer numbers

`~ !`
Bit-wise NOT, logical NOT. Same precedence as unary minus and unary plus.

`\%`
Remainder. Same precedence as multiply and divide.

`<< >>`
Left and right (bit) shift. Precedence lower than add and subtract.

`&`
Bit-wise AND. Each pair of bits is subjected to the logical operation. `\^`

Bit-wise exclusive OR. `|`

Bit-wise OR.

Note: When applied to integers, the division and remainder operators can be considered to partition the number line into a sequence of equal-sized adjacent non-overlapping pieces where each piece is the size of the divisor; the division result identifies which piece the divisor lay within, and the remainder result identifies where within that piece the divisor lay. A consequence of this is that the result of `-57 / 10` is always `-6`, and the result of `-57 \% 10` is always `3`.

Operations on strings

`== != < > <= >= <`
Relational operators: is equal, not equal, less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. With either one or two strings as operands these operations use string comparison (alphabetical-lexicorgraphical comparison, using the ASCII/UNICODE table).

`eq ne in ni`
Compare two strings for equality (`eq`) or inequality (`ne`) and two operators for checking if a string is contained in a list (`in`) or not (`ni`). These operators all return 1 (true) or 0 (false). Using these operators ensures that the operands are regarded exclusively as strings (and lists), not as possible numbers.

Note the difference between `==` and `eq` (analoguously `!=` and `ne`):

```
% expr { "9" == "9.0"} ;# Operands can be interpreted as numbers
1
% expr { "9" eq "9.0"} ;# No attempt is made to convert the operands to numbers!
0
```

Logical operations

Expressions can be combined to form more complicated expressions:

`&\&`
Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).

`||`
Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).

`x?y:z`
If-then-else. If x evaluates to non-zero, then the result is the value of y. Otherwise the result is the value of z. The x operand must have a numeric value:

You can influence the order of evaluation using parentheses: `expr {3+4*5}` gives `23` and `expr {(3+4)*5}` gives `35`.

For example:

```
% set x 1
1
% expr { $x % 2 ? "Odd" : "Even" }
Odd
% set y 3
% expr {$x > 1 || ($x < 2 && $y == 3)}
1
```

Math functions

Tcl supports the following mathematical functions in expressions:

```
abs acos asin atan
atan2 bool ceil cos
cosh double entier exp
floor fmod hypot int
isqrt log log10 max
min pow rand round
sin sinh sqrt srand
tan tanh wide
```

Besides these functions, you can also apply commands within an expression. For instance:

```
% set x 1
1
% set w "Abcdef"
Abcdef
% expr { [string length $w]-2*$x }
4
```

It is even possible to define your own math functions, though this is a somewhat advanced subject and requires some understanding of namespaces.

The above mathematical functions and operators also exist as independent commands:

```
% expr {sqrt(4)}
2.0
% ::tcl::mathfunc::sqrt 4 ;# sqrt lives in a separate "namespace", "::tcl::mathfunc"
2.0
```

Type conversions

Tcl supports the following functions to convert from one representation of a number to another:

```
double int wide entier
```

- `double()` converts a number to a double-precision floating-point number.
- `int()` converts a number to an ordinary integer number (by *truncating* the decimal part).
- `wide()` converts a number to a so-called wide integer number (these numbers have a larger range).
- `entier()` coerces a number to an integer of appropriate size to hold it without truncation. This might return the sameas int() or wide() or an integer of arbitrary size (in Tcl 8.5 and above).

The *next lesson* (Section 1.9) explains the various types of numbers in more detail.

Examples

Some mathematical expressions:

```
set X 100
set Y 256
set Z [expr {$Y + $X}]
set Z_LABEL "$Y plus $X is "

puts "$Z_LABEL $Z"
puts "The square root of $Y is [expr { sqrt($Y) }]\n"

puts "Because of the precedence rules \"5 + -3 * 4\" is: [expr {-3 * 4 + 5}]"
puts "Because of the parentheses \"(5 + -3) * 4\" is: [expr {(5 + -3) * 4}]"
```

Result:

```
256 plus 100 is 356
The square root of 256 is 16.0

Because of the precedence rules "5 + -3 * 4" is: -7
Because of the parentheses "(5 + -3) * 4" is: 8
```

```
set A 3
set B 4
puts "The hypotenuse of a triangle: [expr {hypot($A,$B)}]"

#
# The trigonometric functions work with radians ...
#
set pi6 [expr {3.1415926/6.0}]
puts "The sine and cosine of pi/6: [expr {sin($pi6)}] [expr {cos($pi6)}]"
```

Result:

```
The hypotenuse of a triangle: 5.0
The sine and cosine of pi/6: 0.49999999226497965 0.8660254082502546
```

```
#
# Working with arrays
#
set a(1) 10
set a(2) 7
set a(3) 17
set b 2
puts "Sum: [expr {$a(1)+$a($b)}]"
```

Result: Sum: 17

Bracing your expressions

Consider the following commands:

```
% set userinput {[puts DANGER!]}
[puts DANGER!]
% expr $userinput == 1
DANGER!
0
% expr {$userinput == 1}
0
```

In the first example, the code contained in the user-supplied input is evaluated, whereas in the second the braces prevent this potential danger. As a general rule, *always surround expressions with braces*, whether using `expr` directly or some other command that takes an expression (such as `if` or `while`).

Numbers with a leading zero

Beware of leading zeros: 0700 is not interpreted as the decimal number 700 (seven hundred), but as the *octal* number 700 = 7*8*8 = 448 (decimal).

Worse, if the number contains a digit 8 or 9 an error results:

```
% expr {0900+1}
expected integer but got "0900" (looks like invalid octal number)
```

Octal numbers are in fact a relic of the past, when such number formats were much more common.

If you need to read in decimal data that might have leading zeros, then use the `scan` command to properly convert them into numbers without the above problems.

1.9 Computers and numbers

If you are new to programming, then this lesson may contain some surprising information. But even if you are used to writing programs, computers can do unexpected things with numbers. The purpose of this lesson is to shed some light on some of the mysteries and quirks you can encounter. These mysteries exist independently of the programming language, though one programming language may be better at isolating you from them than another. The problem is that computers do not deal with the numbers *we* are used and in the way we are used to.

```
% expr {1/6}
0
% expr {1/6.0}
0.16666666666666666
```

The difference is the decimal point in the second expression.

Tcl's strategy

Tcl uses a simple but efficient strategy to decide what kind of numbers to use for the computations:

- If you add, subtract, multiply and divide two *integer numbers*, then the result is an integer. If the result fits within the range you have the exact answer. If not, you end up with something that appears to be completely wrong. (*Note:* not too long ago, floating-point computations were much more time-consuming than integer computations. And most computers do not warn about integer results outside the range, because that is too time-consuming too: a computer typically uses lots of such operations, most of which do fit into the designated range.)
- If you add, subtract, multiply and divide an integer number and a *floating-point number*, then the integer number is first converted to a floating-point number with the same value and then the computation is done, resulting in a floating-point number.

Floating-point computations are quite complex, and the current(IEEE) standard prescribes what should happen in minute detail. One such detail is that results outside the proper ranges are reported. Tcl catches these and displays a warning:

```
% # Compute 1.0e+300/1.0-300
% puts [expr {1.0e300/1.0e-300}]
floating-point value too large to represent
```

What are those mysteries and quirks?

Now some of the mysteries you can find yourself involved in. Run the following scripts:

```
#
# Division
#
puts "1/2 is [expr {1/2}]"
puts "-1/2 is [expr {-1/2}]"
puts "1/2 is [expr {1./2}]"
puts "1/3 is [expr {1./3}]"
puts "1/3 is [expr {double(1)/3}]"
```

The first two computations have the surprising result: 0 and -1. That is because the result is an *integer number* and the mathematically exact results 1/2 and -1/2 are not. If you interested in the details of how Tcl works, the outcome *q* is determined as follows:

```
a = q * b + r
0 <= |r| < |b|
r has the same sign as q
```

Here are some examples with floating-point numbers:

```
puts "1/2 is [expr {1./2}]"
puts "1/3 is [expr {1./3}]"

set a [expr {1.0/3.0}]
puts "3*(1/3) is [expr {3.0*$a}]"

set b [expr {10.0/3.0}]
puts "3*(10/3) is [expr {3.0*$b}]"

set c [expr {10.0/3.0}]
set d [expr {2.0/3.0}]
puts "(10.0/3.0) / (2.0/3.0) is [expr {$c/$d}]"

set e [expr {1.0/10.0}]
puts "1.2 / 0.1 is [expr {1.2/$e}]"
```

While many of the above computations give the result you would expect, note however the last decimals, the last two do not give exactly 5 and 12! This is because computers can only deal with numbers with a limited precision: floating-point numbers are *not* our mathematical real numbers.

Somewhat unexpectedly, *1/10* also gives problems. 1.2/0.1 results in 11.999999999999998, not 12. That is an example of a very nasty aspect of most computers and programming languages today: they do not work with ordinary decimal fractions, but with binary fractions. So, 0.5 can be represented exactly, but 0.1 can not.

Some practical consequences

- The fact that floating-point numbers are not ordinary decimal or real numbers and the actual way computers deal with floating-point numbers, has a number of consequences: Results obtained on one computer may not *exactly* match the results on another computer. Usually the differences are small, but if you have a lot of computations, they can add up!
- Whenever you convert from floating-point numbers to integer numbers, for instance when determining the labels for a graph (the range is 0 to 1.2 and you want a step size of 0.1), you need to be careful:

```
#
# The wrong way
#
set number [expr {int(1.2/0.1)}] ;# Force an integer -
```

```
                              ;# accidentally number = 11

for { set i 0 } { $i <= $number } { incr i } {
   set x [expr {$i*0.1}]
   ... create label $x
}



#
# A right way - note the limit
#
set x 0.0
set delta 0.1
while { $x < 1.2+0.5*$delta } {
   ... create label $x
   set x [expr {$x + $delta}]
}
```

- If you want to do *financial* computations, take care: there are specific standards for doing such computations that unfortunately depend on the country where they are used - the US standard is slightly different from the European standard.
- Transcendental functions, like sin() and exp() are not standardised at all. The outcome could differ in one or more decimals from one computer to the next. So, if you want to be absolutely certain that the mathematical number ”pi” is a specific value, use that value and do not rely on formulae like these:

```
#
# Two different estimates of "pi"
#
set pi1 [expr {4.0*atan(1.0)}]
set pi2 [expr {6.0*asin(0.5)}]
puts [expr {$pi1-$pi2}]
-4.4408920985006262e-016
```

Chapter 2

Flow control

2.1 Numeric Comparisons 101 - if

Like most languages, Tcl supports an `if` command. The syntax is:

```
if {expr1} ?then? {
    body1
} elseif {expr2} ?then? {
    body2
} elseif {
    ...
} ?else {
    bodyN
}?
```

The words `then` and `else` are optional, although usually `then` is left out and `else` is used.

The test expression following `if` should return a value that can be interpreted as representing "true" or "false": If the test expression returns a string

	False	True
a numeric value	0	all others
yes/no	no	yes
true/false	false	true

"yes"/"no" or "true"/"false", the case of the return is not checked. True/FALSE or YeS/nO are legitimate returns.

If the test expression evaluates to True, then `body1` will be executed.

If the test expression evaluates to False, then the word after `body1` will be examined. If the next word is `elseif`, then the next test expression will be tested as a condition. If the next word is `else` then the final `body` will be evaluated as a command.

The test expression following the word `if` is evaluated in the same manner as in the `expr` command.

The test expression following `if` should be enclosed within braces. This causes the expression to be evaluated within the `if` command.

Note: as was explained in the *lesson on the expr command* (Section 1.8), you should always use braces around expressions.

Example

```
set x 1

if {$x == 2} {puts "$x is 2"} else {puts "$x is not 2"}

if {$x != 1} {
    puts "$x is != 1"
} else {
    puts "$x is 1"
}
```

Result:

```
1 is not 2
1 is 1
```

2.2 Textual Comparison - switch

The `switch` command allows you to choose one of several options in your code. It is similar to `switch` in C, except that it is more flexible, because you can switch on strings, instead of just integers. The string will be compared to a set of patterns, and when a pattern matches the string, the code associated with that pattern will be evaluated.

It's a good idea to use the `switch` command when you want to match a variable against several possible values, and don't want to do a long series of `if... elseif ... elseif` statements.

The syntax of the command is:

```
   switch ?options? string {
       pattern1 {
           body1
       }
       ?pattern2 {
           body2
       }?
       ...
       ?patternN {
           bodyN
       }?
   }
```

`string` is the string that you wish to test, and `pattern1, pattern2, etc` are the patterns that the string will be compared to. If `string` matches a pattern, then the code within the `body` associated with that pattern will be executed. The return value of the `body` will be returned as the return value of the switch statement. Only one pattern will be matched.

If the last `pattern` argument is the string `default`, that pattern will match any string. This guarantees that some set of code will be executed no matter what the contents of `string` are.

If there is no `default` argument, and none of the `patterns` match `string`, then the `switch` command will return an empty string.

The `options` can be used to change the interpretation of the patterns. By default "glob" style pattern matching is used, where an asterisk (*) matches any number of characters, that is a pattern "lesson*" matches "lesson", "lessons", "lession 2" etc.

Example

```
set x "ONE"
set y 1
set z ONE

# Note that patterns are not subject to substitutions within braces
switch $x {
    "$z" {
        set y1 [expr {$y+1}]
        puts "MATCH \$z. $y + $z is $y1"
    }
    ONE {
        set y1 [expr {$y+1}]
        puts "MATCH ONE. $y + one is $y1"
    }
    TWO {
        set y1 [expr {$y+2}]
        puts "MATCH TWO. $y + two is $y1"
    }
    THREE {
        set y1 [expr {$y+3}]
        puts "MATCH THREE. $y + three is $y1"
    }
    default {
        puts "$x is NOT A MATCH"
    }
}
```

Result:

```
MATCH ONE. 1 + one is 2
```

2.3 Looping 101 - While loop

Tcl includes three commands for looping, the `while`, `for` and `foreach` commands. Like the `if` statement, they evaluate their test the same way that the `expr` does. In this lesson we discuss the `while` command, and in the next lesson, the `for` command. The last command will be treated together with lists, as it iterates over one or more lists.

In many circumstances where one of the commands `while` or `for` can be used, the other can be used as well.

```
while test {
    body
}
```

The `while` command evaluates `test` as an expression. If `test` is true, the code in `body` is executed. After the code in `body` has been executed, `test` is evaluated again.

A `continue` statement within `body` will cause the rest to be skipped and execution continues with the next iteration. A `break` within `body` will break out of the while loop, and execution will continue after the closing brace.

In Tcl **everything** is a command, and everything goes through the same substitution phase. For this reason, the `test` must be placed within braces. If `test` is placed within quotes, the substitution phase will replace any variables with their current value, and will pass that test to the `while` command to evaluate, and since the test has only numbers, it will always evaluate the same, quite probably leading to an endless loop!

Examples

```
set x 1

# This is a normal way to write a Tcl while loop.

while {$x < 5} {
    puts "x is $x"
    set x [expr {$x + 1}]
}

puts "exited first loop with X equal to $x\n"
```

Result:

```
x is 1
x is 2
x is 3
x is 4
exited first loop with X equal to 5
```

Look at the next loop. If it weren't for the break command in the second loop, it would loop forever.

```
# The next example shows the difference between ".." and {...}
# How many times does the following loop run? Why does it not
# print on each pass?

set x 0
```

```
while "$x < 5" {
    set x [expr {$x + 1}]
    if {$x > 7} break
    if "$x > 3" continue
    puts "x is $x"
}

puts "exited second loop with X equal to $x"
```

Result:

```
x is 1
x is 2
x is 3
exited second loop with X equal to 8
```

2.4 Looping 102 - For and incr

Tcl supports a loop construct similar to the `for` loop in C. The `for` command in Tcl takes four arguments; an initialization, a test, an increment, and the body of code to evaluate on each pass through the loop. The syntax for the `for` command is:

```
   for start test next body
```

During evaluation of the `for` command, the `start` code is evaluated once, before any other arguments are evaluated. After the start code has been evaluated, the `test` is evaluated. If the `test` evaluates to true, then the `body` is evaluated, and finally, the `next` argument is evaluated. After evaluating the `next` argument, the interpreter loops back to the `test`, and repeats the process. If the `test` evaluates as false, then the loop will exit immediately.

`start` is the initialization portion of the command. It is usually used to initialize the iteration variable, but can contain any code that you wish to execute before the loop starts.

The `test` argument is evaluated as an expression, just as with the `expr` `while` and `if` commands.

`next` is commonly an incrementing command, but may contain any command which the Tcl interpreter can evaluate.

`body` is the body of code to execute.

When braces are used for grouping, the newline is not treated as the end of a Tcl command. This makes it simpler to write multiple line commands. However, the opening brace **must** be on the line with the `for` command, or the Tcl interpreter will treat the close of the `next` brace as the end of the command, and you will get an error. This is different than other languages like C or Perl, where it doesn't matter where you place your braces.

Within the `body` code, the commands `break` and `continue` may be used just as they are used with the `while` command. When a `break` is encountered, the loop exits immediately. When a `continue` is encountered, evaluation of the `body` ceases, and the next iteration is started (if there is one left).

Because incrementing the iteration variable is so common, Tcl has a special command for this:

```
   incr varName ?increment?
```

This command adds the value in the second argument to the variable named in the first argument. If no value is given for the second argument, it defaults to 1.

Example

```
for {set i 0} {$i < 10} {incr i} {
    puts "I inside first loop: $i"
}

for {set i 3} {$i < 2} {incr i} {
    puts "I inside second loop: $i"
}

puts "Start"
set i 0
while {$i < 10} {
    puts "I inside third loop: $i"
    incr i
    puts "I after incr: $i"
}

set i 0
incr i
# This is equivalent to:
set i [expr {$i + 1}]
```

Result:

```
I inside first loop: 0
I inside first loop: 1
I inside first loop: 2
I inside first loop: 3
I inside first loop: 4
I inside first loop: 5
I inside first loop: 6
I inside first loop: 7
I inside first loop: 8
I inside first loop: 9
Start
I inside third loop: 0
I after incr: 1
I inside third loop: 1
I after incr: 2
I inside third loop: 2
I after incr: 3
```

```
I inside third loop: 3
I after incr: 4
I inside third loop: 4
I after incr: 5
I inside third loop: 5
I after incr: 6
I inside third loop: 6
I after incr: 7
I inside third loop: 7
I after incr: 8
I inside third loop: 8
I after incr: 9
I inside third loop: 9
I after incr: 10
```

2.5 Adding new commands to Tcl - proc

In Tcl there is actually no distinction between commands (often known as 'functions' in other languages) and "syntax". There are no reserved words (like if and while) as exist in C, Java, Python, Perl, etc... When the Tcl interpreter starts up there is a list of known commands that the interpreter uses to parse a line. These commands include `while, for, set, puts,` and so on. They are, however, still just regular Tcl commands that obey the same syntax rules as all Tcl commands, both built-in, and those that you create yourself with the `proc` command.

The `proc` command creates a new command. The syntax for the `proc` command is:

```
   proc name arguments body
```

When `proc` is evaluated, it creates a new command with name `name` that takes arguments `args`. When the procedure `name` is called, it then runs the code contained in `body`.

`arguments` is a list of arguments which will be passed to `name`. When `name` is invoked, local variables with these names will be created, and the values to be passed to `name` will be copied to the local variables.

The value that the `body` of a proc returns can be defined with the `return` command. The `return` command will return its argument to the calling program. If there is no return, then `body` will return to the caller when the last of its commands has been executed. The return value of the last command becomes the return value of the procedure.

Example

```
proc sum {arg1 arg2} {
    set x [expr {$arg1 + $arg2}];
    return $x
}
```

```
puts " The sum of 2 + 3 is: [sum 2 3]\n\n"
```

Result:

```
 The sum of 2 + 3 is: 5
```

If you make a mistake in the call, Tcl will write an error message:

```
puts " The sum of 2 + 3 is: [sum 2]\n\n"
```

gives:

```
wrong # args: should be "sum arg1 arg2"
    while executing
"sum 2"
    invoked from within
"puts " The sum of 2 + 3 is: [sum 2]\n\n""
    (file "xx.tcl" line 6)
```

Result: As an aside, it is possible to redefine any builtin command or procedure. This is useful for instance for debugging purposes, although there are other means for this as well. As a simple demonstration:

```
proc for {a b c} {
    puts "The for command has been replaced by a puts";
    puts "The arguments were: $a\n$b\n$c\n"
}

for {set i 1} {$i < 10} {incr i}
```

2.6 Variations in proc arguments and return values

A proc can be defined with a set number of required arguments (as was done with `sum` in the previous lesson, or it can have a variable number of arguments. An argument can also be defined to have a default value.

Variables can be defined with a default value by placing the variable name and the default within braces within `args`. For example:

```
proc justdoit {a {b 1} {c -1}} {
    ...
}
```

Since there are default arguments for the `b` and `c` variables, you could call the procedure one of three ways: `justdoit 10`, which would set `a` to 10, and leave `b` set to its default 1, and `c` at -1. `justdoit 10 20` would likewise set `b` to 20, and leave `c` to its default. Or call it with all three parameters set to avoid any defaults.

A proc will accept a variable number of arguments if the last declared argument is the word `args`. If the last argument to a proc argument list is `args`, then any arguments that aren't already assigned to previous variables will be assigned to `args`.

```
proc show_a_list {args} {
    set n 0
    foreach arg $args {
        puts "Argument $n: $arg"
        incr n
    }
}

show_a_list A B C D
puts ""
show_a_list E F
```

results in:

```
Argument 0: A
Argument 1: B
Argument 2: C
Argument 3: D

Argument 0: E
Argument 1: F
```

Note that if there is a variable other than `args` after a variable with a default, then the default will never be used. For example, if you declare a proc such as:

```
proc function { a {b 1} c} {...}
```

you will always have to call it with 3 arguments.

Tcl assigns values to a proc's variables in the order that they are listed in the command. If you provide 2 arguments when you call `function` they will be assigned to `a` and `b`, and Tcl will generate an error because `c` is undefined.

You can, however, declare other arguments that may not have values as coming after an argument with a default value. For example, this is valid:

```
proc example {required {default1 a} {default2 b} args} {...}
```

In this case, `example` requires one argument, which will be assigned to the variable `required`. If there are two arguments, the second arg will be assigned to `default1`. If there are 3 arguments, the first will be assigned to `required`, the second to `default1`, and the third to `default2`. If `example` is called with more than 3 arguments, all the arguments after the third will be assigned to `args`.

Example

The `example` procedure below is defined with three arguments. At least one argument *must* be present when `example` is called. The second argument can be left out, and in that case it will default to an empty string. By declaring `args` as the last argument, `example` can take a variable number of arguments.

Also note the use of the `return` statement to explicitly return a particular result.

```
proc example {first {second ""} args} {
    if {$second eq ""} {
        puts "There is only one argument and it is: $first"
        return 1
    } else {
        if {$args eq ""} {
            puts "There are two arguments - $first and $second"
            return 2
        } else {
            puts "There are many arguments - $first and $second and $args"
            return "many"
        }
    }
}

set count1 [example ONE]
set count2 [example ONE TWO]
set count3 [example ONE TWO THREE ]
set count4 [example ONE TWO THREE FOUR]

puts "The example was called with $count1, $count2, $count3, and $count4 Arguments"
```

Result:

```
There is only one argument and it is: ONE
There are two arguments - ONE and TWO
There are many arguments - ONE and TWO and THREE
There are many arguments - ONE and TWO and THREE FOUR
The example was called with 1, 2, many, and many Arguments
```

2.7 Variable scope - global and upvar

Tcl evaluates variables within a *scope* delineated by procs, namespaces (see *Building reusable libraries - packages and namespaces* (Section 6.2)), and at the topmost level, the `global` scope.

The scope in which a variable will be evaluated can be changed with the `global` and `upvar` commands.

The `global` command will cause a variable in a local scope (inside a procedure) to refer to the global variable of that name.

The `upvar` command is similar. It ”ties” the name of a variable in the current scope to a variable in a different scope. This is commonly used to simulate pass-by-reference to procs.

You might also encounter the `variable` command in others’ Tcl code. It is part of the namespace system and is discussed in detail in the *chapter on namespaces* (Section 6.2).

Normally, Tcl uses a type of ”garbage collection” called reference counting in order to automatically clean up variables when they are not used anymore, such as when they go ”out of scope” at the end of a procedure, so that you don't have to keep track of them yourself. It is also possible to explicitly unset them with the aptly named `unset` command.

The syntax for `upvar` is:

```
upvar ?level? otherVar1 myVar1 ?otherVar2 myVar2? ... ?otherVarN myVarN?
```

The `upvar` command causes `myVar1` to become a reference to `otherVar1`, and `myVar2` to become a reference to `otherVar2`, etc. The `otherVar` variable is declared to be at `level` relative to the current procedure. By default `level` is 1, the next level up, though it is best to always set it explicitly.

If a number is used for the `level`, then level references that many levels up the stack from the current level.

If the `level` number is preceded by a `\#` symbol, then it references that many levels down from the global scope. If `level` is `\#0`, then the reference is to a variable at the global level.

If you are using upvar with anything except #0 or 1, you are most likely asking for trouble, unless you really know what you're doing.

You should avoid using global variables if possible. If you have a lot of globals, you should reconsider the design of your program.

Note that since there is only one global space it is surprisingly easy to have name conflicts if you are importing other people's code and aren't careful. It is recommended that you start global variables with an identifiable prefix to help avoid unexpected conflicts.

Example

The `global` command can be used for data you need to share:

```
global logFile
set logFile [open "log.out" w]

proc writeLog {data} {
    global logFile

    puts $logFile "LOG: $data"
}
```

This is a simplistic way to provide logging information.

The `upvar` command can be used to pass variables "by reference":

```
proc SetPositive {variable value } {
    upvar 1 $variable myvar
    if {$value < 0} {
        set myvar [expr {-$value}]
    } else {
        set myvar $value
    }
    return $myvar

    # Or more concisely:
    # set myvar [expr {abs($myvar)}]
}
```

```
SetPositive x 5
SetPositive y -5

puts "X : $x Y: $y\n"
```

resulting in:

```
X : 5 Y: 5
```

Chapter 3

Data types

3.1 Tcl Data Structures 101 - The list

The list is the basic Tcl data structure. A list is simply an ordered collection of stuff; numbers, words, strings, or other lists. Even commands in Tcl are just lists in which the first list entry is the name of a proc, and subsequent members of the list are the arguments to the proc.

Lists can be created in several ways:

- by setting a variable to be a list of values

  ```
  set lst {{item 1} {item 2} {item 3}}
  ```

- with the `split` command:

  ```
  set lst [split "item 1.item 2.item 3" "."]
  ```

- with the `list` command.

  ```
  set lst [list "item 1" "item 2" "item 3"]
  ```

An individual list member can be accessed with the `lindex` command.

The brief description of these commands is:

`list ?arg1? ?arg2? ... ?argN?`
makes a list of the arguments

`split string ?splitChars?`
splits the `string` into a list of items wherever the `splitChars` occur in the code. `SplitChars` defaults to being whitespace. Note that if there are two or more `splitChars` then each one will be used individually to split the string. In other words: `split  "1234567" "36"` would return the following list: 12 45 7.

`lindex list index`
Returns the `index`'th item from the list. **Note:** lists start from 0, not 1, so the first item is at index 0, the second item is at index 1, and so on.

`llength list`
Returns the number of elements in a list.

The items in list can be iterated through using the `foreach` command:

```
    foreach varName list body
```

The `foreach` command will execute the `body` code one time for each list item in `list`. On each pass, `varName` will contain the value of the next `list` item.

In reality, the above form of `foreach` is the simple form, but the command is quite powerful. It will allow you to take more than one variable at a time from the list: `foreach {a b} \$listofpairs { ... }`. You can even take a variable at a time from multiple lists! For example:

```
    foreach a $listOfA b $listOfB {
        ...
    }
```

or:

```
    foreach {a b} $listOfAB {
        ...
    }
```

Furthermore, lists can be nested:

```
    set nestedList {
        {1 2 3}
        {A B C D}
        {xyz 33 1A}
    }

    puts "The last element from the second sublist is: [lindex $nestedList 1 end]"
```

producing:

```
The last element from the second sublist is: D
```

Examples

```
set x "a b c d e f g h"
puts "Item at index 2 of the list {$x} is: [lindex $x 2]\n"

set y [split 7/4/1776 "/"]
puts "We celebrate on the [lindex $y 1]'th day of the [lindex $y 0]'th month\n"

set z [list puts "arg 2 is $y" ]
puts "A command resembles: $z\n"

set i 0
foreach j $x {
    puts "$j is item number $i in list x"
    incr i
}
set i 0
foreach {a b} $x {
    puts "Pair $i in list x: $a, $b"
    incr i
}
```

Result:

```
Item at index 2 of the list {a b c d e f g h} is: c

We celebrate on the 4'th day of the 7'th month

A command resembles: puts {arg 2 is 7 4 1776}

a is item number 0 in list x
b is item number 1 in list x
c is item number 2 in list x
d is item number 3 in list x
e is item number 4 in list x
f is item number 5 in list x
g is item number 6 in list x
h is item number 7 in list x
Pair 0 in list x: a, b
Pair 1 in list x: c, d
Pair 2 in list x: e, f
Pair 3 in list x: g, h
```

3.2 Adding and Deleting members of a list

The commands for adding and deleting list members are:

`concat ?arg1 arg2 ... argn?`
Concatenates the `args` into a single list. It also eliminates leading and trailing spaces in the args and adds a single separator space between args. The `args` to `concat` may be either individual elements, or lists. If an `arg` is already a list, the contents of that list is concatenated with the other `args`.

`lappend varName ?arg1 arg2 ... argn?`
Appends the `args` to the list in variable `varName` treating each `arg` as a list element.

`linsert listValue index arg1 ?arg2 ... argn?`
Returns a new list with the new list elements inserted just before the `index`th element of `listValue`. Each element argument will become a separate element of the new list. If index is less than or equal to zero, then the new elements are inserted at the beginning of the list. If index has the value `end` , or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

`lreplace \verb`**listValue first last ?arg1 ... argn¿**
Returns a new list with N elements of `listName` replaced by the `args`. If `first` is less than or equal to 0, lreplace starts replacing from the first element of the list. If `first` is greater than the end of the list, or the word `end`, then lreplace behaves like lappend. If there are fewer `args` than the number of positions between `first` and `last`, then the positions for which there are no `args` are deleted.

`lset varName index newValue`
The `lset` command can be used to set elements of a list directly, instead of using `lreplace`. With nested lists, the index can consist of several indices, one for each level: `lset varName 1 2 3 "Hello, world"`

Lists in Tcl are the right data structure to use when you have an arbitrary number of things, and you'd like to access them according to their order in the list. In C, you would use an array. In Tcl, arrays are associative arrays - hash tables, as you'll see in the coming sections. If you want to have a collection of things, and refer to the Nth thing (give me the 10th element in this group of numbers), or go through them in order via `foreach`. An alternative is the *dictionary or* `dict`‘ (Section 3.14).

Take a look at the example code, and pay special attention to the way that sets of characters are grouped into single list elements.

Example

```
set b [list a b {c d e} {f {g h}}]
puts "Treated as a list: $b\n"

set b [split "a b {c d e} {f {g h}}"]
puts "Transformed by split: $b\n"

set a [concat a b {c d e} {f {g h}}]
puts "Concated: $a\n"

lappend a {ij K lm} ;# Note: {ij K lm} is a single element
puts "After lappending: $a\n"

set b [linsert $a 3 "1 2 3"] ;# "1 2 3" is a single element
puts "After linsert at position 3: $b\n"

set b [lreplace $b 3 5 "AA" "BB"]
puts "After lreplacing 3 positions with 2 values at position 3: $b\n"
```

Result:

```
Treated as a list: a b {c d e} {f {g h}}

Transformed by split: a b \{c d e\} \{f \{g h\}\}

Concated: a b c d e f {g h}
```

```
After lappending: a b c d e f {g h} {ij K lm}

After linsert at position 3: a b c {1 2 3} d e f {g h} {ij K lm}

After lreplacing 3 positions with 2 values at position 3: a b c AA BB f {g h} {ij K lm}
```

3.3 More list commands - lsearch, lsort, lrange

Lists can be searched with the `lsearch` command, sorted with the `lsort` command, and a range of list entries can be extracted with the `lrange` command.

`lsearch ?options? list pattern`
Searches `list` for an entry that matches `pattern`, and returns the index for the first match, or a -1 if there is no match. By default, `lsearch` uses "glob" patterns for matching. See the section on ¡a href="Tcl16a.html"¿globbing¡/a¿.

`lsort ?options? list`
Sorts `list` and returns a new list in the sorted order. By default, it sorts the list into alphabetic order. Note that this command returns the sorted list as a result, instead of sorting the list in place. If you have a list in a variable, the way to sort it is like so: `set lst [lsort \$lst]`

`lrange list first last`
Returns a list composed of the `first` through `last` entries in the list. If `first` is less than or equal to 0, it is treated as the first list element. If `last` is `end` or a value greater than the number of elements in the list, it is treated as the end. If `first` is greater than `last` then an empty list is returned.

While the options are not discussed here, they make these commands very powerful.

Example

```
set list [list {Washington 1789} {Adams 1797} {Jefferson 1801} \
              {Madison 1809} {Monroe 1817} {Adams 1825} ]

set x [lsearch $list Washington*]
set y [lsearch $list Madison*]
incr x
incr y -1 ;# Set range to be not-inclusive

set subsetlist [lrange $list $x $y]

puts "The following presidents served between Washington and Madison"
foreach item $subsetlist {
    puts "Starting in [lindex $item 1]: President [lindex $item 0] "
}

set x [lsearch $list Madison*]
set srtlist [lsort $list]
set y [lsearch $srtlist Madison*]

puts "\n$x Presidents came before Madison chronologically"
puts "$y Presidents came before Madison alphabetically"
```

Result:

```
The following presidents served between Washington and Madison
Starting in 1797: President Adams
Starting in 1801: President Jefferson

3 Presidents came before Madison chronologically
3 Presidents came before Madison alphabetically
```

3.4 Simple pattern matching - ”globbing”

By default, lsearch uses the ”globbing” method of finding a match. Globbing is the wildcarding technique that most Unix shells use.

globbing wildcards are:

*****
Matches any quantity of any character

?
Matches one occurrence of any character

X
The backslash escapes a special character in globbing just the way it does in Tcl substitutions. Using the backslash lets you use glob to match a `*` or `?`.

[...]
Matches one occurrence of any character within the brackets. A range of characters can be matched by using a range between the brackets. For example, [a-z] will match any lower case letter.

There is also a `glob` command that you will see in later sections that uses glob pattern matching in directories, and returns a list of the matching files.

Example

```
# Matches
string match f* foo

# Matches
string match f?? foo

# Doesn’t match
```

```
string match f foo

# Returns a big list of files on my Debian system.
set bins [glob /usr/bin/*]
```

3.5 String Subcommands - length index range

Tcl commands often have ”subcommands”. The `string` command is an example of one of these. The `string` command treats its first argument as a subcommand. Utilizing subcommands is a good way to make one command do multiple things without using cryptic names. For instance, Tcl has `string length` instead of, say, `slength`.

This lesson covers these string subcommands:

`string length str`
Returns the length of `str`.

`string index str i`
Returns the `i`th character from `str`.

`string range str first last`
Returns a string composed of the characters from `first` to `last` taken from `str`.

Example

```
set string "this is my test string"

puts "There are [string length $string] characters in \"$string\""

puts "[string index $string 1] is the second character in \"$string\""

puts "\"[string range $string 5 10]\" are characters between the 5’th and 10’th"
```

Result:

```
There are 22 characters in "this is my test string"
h is the second character in "this is my test string"
"is my " are characters between the 5’th and 10’th
```

3.6 String comparisons - compare match first last wordend

There are 6 string subcommands that do pattern and string matching. These are relatively fast operations, certainly faster than regular expressions, albeit less powerful.

`string compare string1 string2`
Compares `string1` to `string2` and returns: ¡ul¿ ¡li¿-1 ..... If `string1` is less than `string2` ¡li¿ 0 .... If `string1` is equal to `string2` ¡li¿ 1 .... If `string1` is greater than `string2` ¡/ul¿ These comparisons are done alphabetically, not numerically - in other words ”a” is less than ”b”, and ”10” is less than ”2”.

`string first string1 string2`
Returns the index of the character in `string1` that starts the first match to `string2`, or -1 if there is no match.

`string last string1 string2`
Returns the index of the character in `string1` that starts the last match to `string2`, or -1 if there is no match.

`string wordend str i`
Returns the index of the character just after the last one in the word which contains the `i`’th character of `str`. A word is any contiguous set of letters, numbers or underscore characters, or a single other character.

`string wordstart str i`
Returns the index of the first character in the word that contains the `i`’th character of `str`. A word is any contiguous set of letters, numbers or underscore characters, or a single other character.

`string match pattern str`
Returns 1 if the `pattern` matches `string`. The `pattern` is a *glob* (Section 3.4) style pattern.

Example

```
set fullpath "/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17"
set relativepath "CVS/Entries"
set directorypath "/usr/bin/"

set paths [list $fullpath $relativepath $directorypath]

foreach path $paths {
    set first [string first "/" $path]
    set last [string last "/" $path]

    # Report whether path is absolute or relative

    if {$first != 0} {
        puts "$path is a relative path"
    } else {
        puts "$path is an absolute path"
    }

    # If "/" is not the last character in $path, report the last word.
    # else, remove the last "/", and find the next to last "/", and
    # report the last word.

    incr last
    if {$last != [string length $path]} {
        set name [string range $path $last end]
        puts "The file referenced in $path is $name"
    } else {
        incr last -2;
        set tmp [string range $path 0 $last]
        set last [string last "/" $tmp]
        incr last;
        set name [string range $tmp $last end]
        puts "The final directory in $path is $name"
    }

    # CVS is a directory created by the CVS source code control system.
    #

    if {[string match "*CVS*" $path]} {
        puts "$path is part of the source code control tree"
    }

    # Compare to "a" to determine whether the first char is upper or lower case
    set comparison [string compare $name "a"]
    if {$comparison >= 0} {
        puts "$name starts with a lowercase letter\n"
    } else {
        puts "$name starts with an uppercase letter\n"
    }
}
```

Result:

```
/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17 is an absolute path
The file referenced in /usr/home/clif/TCL_STUFF/TclTutor/Lsn.17 is Lsn.17
Lsn.17 starts with an uppercase letter

CVS/Entries is a relative path
The file referenced in CVS/Entries is Entries
CVS/Entries is part of the source code control tree
Entries starts with an uppercase letter

/usr/bin/ is an absolute path
The final directory in /usr/bin/ is bin
bin starts with a lowercase letter
```

3.7 Modifying Strings - tolower, toupper, trim, format

These are the commands which modify a string. Note that none of these modify the string in place. In all cases a new string is returned.

`string tolower str`
Returns `str` with all the letters converted from upper to lower case.

`string toupper str`
Returns `string` with all the letters converted from lower to upper case.

`string trim str ?trimChars?`
Returns `string` with all occurrences of `trimChars` removed from both ends. By default `trimChars` are whitespace (spaces, tabs, newlines). Note that the characters are not treated as a "block" of characters - in other words, `string trim "davidw" dw` would return the string `avi` and not `davi`.

`string trimleft str ?trimChars?`
Returns `string` with all occurrences of `trimChars` removed from the left. By default `trimChars` are whitespace (spaces, tabs, newlines)

`string trimright str ?trimChars?`
Returns `string` with all occurrences of `trimChars` removed from the right. By default `trimChars` are whitespace (spaces, tabs, newlines)

`format formatString ?arg1 arg2 ... argN?`
Returns a string formatted in the same manner as the ANSI sprintf procedure. FormatString is a description of the formatting to use. The full definition of this protocol is in the format man page. A useful subset of the definition is that formatString consists of literal words, backslash sequences, and % fields. The % fields are strings which start with a % and end with one of:

- s... Data is a string
- d... Data is a decimal integer
- x... Data is a hexadecimal integer
- o... Data is an octal integer
- f... Data is a floating point number

The % may be followed by:

- -... Left justify the data in this field
- +... Right justify the data in this field

The justification value may be followed by a number giving the minimum number of spaces to use for the data.

3.8 Example

```
set upper "THIS IS A STRING IN UPPER CASE LETTERS"
set lower "this is a string in lower case letters"
set trailer "This string has trailing dots ...."
set leader "....This string has leading dots"
set both "((this string is nested in parens )))"

puts "tolower converts this: $upper"
puts " to this: [string tolower $upper]\n"
puts "toupper converts this: $lower"
puts " to this: [string toupper $lower]\n"
puts "trimright converts this: $trailer"
puts " to this: [string trimright $trailer .]\n"
puts "trimleft converts this: $leader"
puts " to this: [string trimleft $leader .]\n"
puts "trim converts this: $both"
puts " to this: [string trim $both "()"]\n"

set labels [format "%-20s %+10s " "Item" "Cost"]
set price1 [format "%-20s %10d Cents Each" "Tomatoes" "30"]
set price2 [format "%-20s %10d Cents Each" "Peppers" "20"]
set price3 [format "%-20s %10d Cents Each" "Onions" "10"]
set price4 [format "%-20s %10.2f per Lb." "Steak" "3.59997"]

puts "\n Example of format:\n"
puts "$labels"
puts "$price1"
puts "$price2"
puts "$price3"
puts "$price4"
```

Result:

```
tolower converts this: THIS IS A STRING IN UPPER CASE LETTERS
             to this: this is a string in upper case letters

toupper converts this: this is a string in lower case letters
             to this: THIS IS A STRING IN LOWER CASE LETTERS

trimright converts this: This string has trailing dots ....
               to this: This string has trailing dots

trimleft converts this: ....This string has leading dots
              to this: This string has leading dots

trim converts this: ((this string is nested in parens )))
          to this: this string is nested in parens
```

```
 Example of format:

Item Cost
Tomatoes 30 Cents Each
Peppers 20 Cents Each
Onions 10 Cents Each
Steak 3.60 per Lb.
```

3.9 Regular Expressions 101

Tcl also supports string operations known as ¡I¿regular expressions¡/I¿ Several commands can access these methods with a -regexp argument, see the detailed documentation for which commands support regular expressions.

There are also two explicit commands for parsing regular expressions.

`regexp ?switches? exp string ?matchVar? ?subMatch1 ... subMatchN?`
Searches `string` for the regular expression `exp`. If a parameter `matchVar` is given, then the substring that matches the regular expression is copied to `matchVar`. If `subMatchN` variables exist, then the parenthetical parts of the matching string are copied to the `subMatch` variables, working from left to right.

`regsub ?switches? exp string subSpec varName`
Searches `string` for substrings that match the regular expression `exp` and replaces them with `subSpec`. The resulting string is copied into `varName`.

Regular expressions can be expressed in just a few rules. ˆ

Matches the beginning of a string **$**

Matches the end of a string **.**

Matches any single character *****

Matches any count (0-n) of the previous character **+**

Matches any count, but at least 1 of the previous character **[...]**

Matches any character of a set of characters **[ˆ..]**

Matches any character *NOT* a member of the set of characters following the ˆ **(...)**

groups a set of characer into a subSpec

Regular expressions are similar to the globbing that was discussed in *lesson 16* (Section 3.3) and *lesson 18* (Section 3.6). The main difference is in the way that sets of matched characters are handled. In globbing the only way to select sets of unknown text is the * symbol. This matches to any quantity of any character.

In regular expression parsing, the * symbol matches zero or more occurrences of the character immediately proceeding the *. For example a* would match a, aaaaa, or a blank string. If the character directly before the * is a set of characters within square brackets, then the * will match any quantity of all of these characters. For example, `[a-c]*` would match `aa`, `abc`, `aabcabc`, or again, an empty string.

The + symbol behaves roughly the same as the *, except that it requires at least one character to match. For example, [a-c]+ would match a, abc, or aabcabc, but not an empty string.

Regular expression parsing is more powerful than globbing. With globbing you can use square brackets to enclose a set of characters any of which will be a match. Regular expression parsing also includes a method of selecting any character not in a set. If the first character after the [is a caret ($\hat{}$), then the regular expression parser will match any character not in the set of characters between the square brackets. A caret can be included in the set of characters to match (or not) by placing it in any position other than the first.

The `regexp` command is similar to the `string match` command in that it matches an `exp` against a string. It is different in that it can match a portion of a string, instead of the entire string, and will place the characters matched into the `matchVar` variable.

If a match is found to the portion of a regular expression enclosed within parentheses, `regexp` will copy the subset of matching characters to the `subSpec` argument. This can be used to parse simple strings.

`regsub` will copy the contents of the string to a new variable, substituting the characters that match exp with the characters in subSpec. If subSpec contains a & or
0, then those characters will be replaced by the characters that matched exp. If the number following a backslash is 1-9, then that backslash sequence will be replaced by the appropriate portion of `exp` that is enclosed within parentheses.

Note that the `exp` argument to `regexp` or `regsub` is processed by the Tcl substitution pass. Therefore almost always the expression should be enclosed in braces to prevent any special processing by Tcl.

Example

```
set sample "Where there is a will, There is a way."

#
# Match the first substring with lowercase letters only
#
set result [regexp {[a-z]+} $sample match]
puts "Result: $result match: $match"

#
# Match the first two words, the first one allows uppercase
set result [regexp {([A-Za-z]+) +([a-z]+)} $sample match sub1 sub2 ]
puts "Result: $result Match: $match 1: $sub1 2: $sub2"

#
# Replace a word
#
```

```
regsub "way" $sample "lawsuit" sample2
puts "New: $sample2"

#
# Use the -all option to count the number of "words"
#
puts "Number of words: [regexp -all {[^ ]+} $sample]"
```

Result:

```
Result: 1 match: here
Result: 1 Match: Where there 1: Where 2: there
New: Where there is a will, There is a lawsuit.
Number of words: 9
```

3.10 More Examples Of Regular Expressions

Regular expressions provide a very powerful method of defining a pattern, but they are a bit awkward to understand and to use properly. So let us examine some more examples in detail.

We start with a simple yet non-trivial example: finding *floating-point numbers* in a line of text. Do not worry: we will keep the problem simpler than it is in its full generality. We only consider numbers like `1.0` and not `1.00e+01`.

How do we *design* our regular expression for this problem? By examining typical examples of the strings we want to match:

- Valid numbers are:

  ```
      1.0, .02, +0., 1, +1, -0.0120
  ```

- Invalid numbers (that is, strings we do not want to recognise as numbers but superficially look like them):

  ```
      -, +., 0.0.1, 0..2, ++1
  ```

- Questionable numbers are:

  ```
      +0000 and 0001
  ```

We will accept them - because they normally are accepted and because excluding them makes our pattern more complicated.

A pattern is beginning to emerge:

- A number can start with a sign (- or +) or with a digit. This can be captured with the expression `[-+]?`, which matches a single ”-”, a single ”+” or nothing.
- A number can have zero or more digits in front of a single period (.) and it can have zero or more digits following the period. Perhaps: `[0-9]*\\.[0-9]*` will do ...

- A number may not contain a period at all. So, revise the previous expression to: `[0-9]*\\.?[0-9]*`

The total expression is:

```
[-+]?[0-9]*\.?[0-9]*
```

At this point we can do three things:

- Try the expression with a bunch of examples like the ones above and see if the proper ones match and the others do not.
- Try to make it look nicer, before we start off testing it. For instance the class of characters "[0-9]" is so common that it has a shortcut, "
  d". So, we could settle for:

  ```
  [-+]?\d*\.?\d*
  ```

instead. Or we could decide that we want to capture the digits before and after the period for special processing:

```
[-+]?([0-9])*\.?([0-9]*)
```

- Or, and that may be a good strategy in general!, we can carefully examine the pattern before we start actually using it.

You see, there is a problem with the above pattern: all the parts are optional, that is, each part can match a null string - no sign, no digits before the period, no period, no digits after the period. In other words: *Our pattern can match an empty string!*

Our questionable numbers, like "+000" will be perfectly acceptable and we (grudgingly) agree. But more surprisingly, the strings "–1" and "A1B2" will be accepted too! Why? Because the pattern can start anywhere in the string, so it would match the substrings "-1" and "1" respectively!

We need to reconsider our pattern - it is too simple, too permissive:

- The character before a minus or a plus, if there is any, can not be another digit, a period or a minus or plus. Let us make it a space or a tab or the beginning of the string: `\^|[ \\t]`. This may look a bit strange, but what it says is: either the beginning of the string (ôutside the squarebrackets) *or* (the vertical bar) a space or tab (remember: the string "
  t" represents the tab character).
- Any sequence of digits before the period (if there is one) is allowed: `[0-9]+\\.?`
- There may be zero digits in front of the period, but then there must be at least one digit behind it: `\\.[0-9]+`
- And of course digits in front and behind the period: `[0-9]+\\.[0-9]+`
- The character after the string (if any) can not be a "+","-" or "." as that would get us into the unacceptable number-like strings: `\$|[\^+-.]` (The dollar sign signifies the end of the string)

Before trying to write down the complete regular expression, let us see what different forms we have:

- No period: `[-+]?[0-9]+`
- A period without digits before it: `[-+]?\\.[0-9]+`
- Digits before a period, and possibly digits after it: `[-+]?[0-9]+\\.[0-9]*`

Now the synthesis:

```
(^|[ \t])([-+]?([0-9]+|\.[0-9]+|[0-9]+\.[0-9]*))($|[^+-.])
```

Or:

```
(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])
```

The parentheses are needed to distinguish the alternatives introduced by the vertical bar and to capture the substring we want to have. Each set of parentheses also defines a substring and this can be put into a separate variable:

```
regexp {.....} $line whole char_before number nosign char_after

#
# Or simply only the recognised number (x's as placeholders, the
# last can be left out
#
regexp {.....} $line x x number
```

Tip: To identify these substrings: just count the opening parentheses from left to right.

If we put it to the test:

```
set pattern {(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])}
set examples {"1.0" " .02" " +0."
              "1" "+1" " -0.0120"
              "+0000" " - " "+."
              "0001" "0..2" "++1"
              "A1.0B" "A1"}
foreach e $examples {
    if { [regexp $pattern $e whole \
              char_before number digits_before_period] } {
        puts ">>$e<<: $number ($whole)"
    } else {
        puts ">>$e<<: Does not contain a valid number"
    }
}
```

the result is:

```
>>1.0<<: 1.0 (1.0)
>> .02<<: .02 ( .02)
>> +0.<<: +0. ( +0.)
>>1<<: 1 (1)
```

```
>>+1<<: +1 (+1)
>> -0.0120<<: -0.0120 ( -0.0120)
>>+0000<<: +0000 (+0000)
>> - <<: Does not contain a valid number
>>+.<<: Does not contain a valid number
>>0001<<: 0001 (0001)
>>0..2<<: Does not contain a valid number
>>++1<<: Does not contain a valid number
>>A1.0B<<: Does not contain a valid number
>>A1<<: Does not contain a valid number
```

So our pattern correctly accepts the strings we intended to be recognised as numbers and rejects the others.

Let us turn to some other patterns now:

- Text enclosed in a string: *This is "quoted text".* If we know the enclosing character in advance (double quotes, " in this case), then `"([\^"])*"` will capture the text inside the double quotes.

Suppose we do not know the enclosing character (it can be " or '). Then:

```
regexp {(["'])[^"']*\1} $string enclosed_string
```

will do it; the
1 is a so-called back-reference to the first captured substring.

- You can use this technique to see if a word occurs twice in the same line of text:

```
set string "Again and again and again ..."
if { [regexp {(\y\w+\y).+\1} $string => word] } {
    puts "The word $word occurs at least twice"
}
```

(The pattern
y matches the beginning or the end of a word and
w+ indicates we want at least one character).

- Suppose you need to check the parentheses in some mathematical expression: `(1+a)/(1-b*x)` for instance. A simple check is counting the open and close parentheses:

```
#
# Use the return value of [regexp] to count the number of
# parentheses ...
#
if { [regexp -all {(} $string] != [regexp -all {)} $string] } {
    puts "Parentheses unbalanced!"
}
```

Of course, this is just a rough check. A better one is to see if at any point while scanning the string there are more close parentheses than open parentheses. We can easily extract the parentheses and put them in a list (the **-inline** option does that):

```
set parens [regexp -inline -all {[()]} $string]
set balance 0 set change("(") 1 ;# This technique saves an if-block :)
set change(")") -1
foreach p $parens {
    incr balance $change($p)
    if { $balance < 0 } {
        puts "Parentheses unbalanced!"
    }
}
if { $balance != 0 } {
    puts "Parentheses unbalanced!"
}
```

Finally: Regular expressions are very powerful, but they have certain theoretical limitations. One of these limitations is that they are not suitable for parsing arbitrarily nested text.

You can experiment with regular expressions using the *VisualRegexp* (`http://wiki.tcl.tk/4086`) or *Visual REGEXP* (`http://wiki.tcl.tk/7992`) applications.

More on the theoretical background and practical use of regular expressions (there is lots to cover!) can be found in the book *Mastering Regular Expressions* (`http://wiki.tcl.tk/127`) by J. Friedl.

3.11 More Quoting Hell - Regular Expressions 102

`regexp ?switches? exp string ?matchVar? ?subMatch1 ... subMatchN?` Searches `string` for the regular expression `exp`. If a parameter `matchVar` is given, then the substring that matches the regular expression is copied to `matchVar`. If `subMatchN` variables exist, then the parenthetical parts of the matching string are copied to the `subMatch` variables, working from left to right. `regsub ?switches? exp string subSpec varName`

Searches `string` for substrings that match the regular expression `exp` and replaces them with `subSpec`. The resulting string is copied into `varName`.

The regular expression (`exp`) in the two regular expression parsing commands is evaluated by the Tcl parser during the Tcl substitution phase. This can provide a great deal of power, and also requires a great deal of care.

These examples show some of the trickier aspects of regular expression evaluation. The fields in each example are discussed in painful detail in the most verbose level.

The points to remember as you read the examples are:

* A left square bracket ([) has meaning to the substitution phase, and to the regular expression parser. * A set of parentheses, a plus sign, and a star have meaning to the regular expression parser, but not the Tcl substitution phase. * A backslash sequence (
n,
t, etc) has meaning to the Tcl substitution phase, but not to the regular expression parser. * A backslash escaped character (

The phase at which a character has meaning affects how many escapes are necessary to match the character you wish to match. An escapecan be either enclosing the phrase in braces, or placing a backslash beforethe escaped character. To pass a left bracket to the regular expression parser to evaluate as arange of characters takes 1 escape. To have the regular expressionparser match a literal left bracket takes 2 escapes (one to escape the bracket in the Tcl substitution phase, and one to escape the bracket inthe regular expression parsing.). If you have the string placed withinquotes, then a backslash that you wish passed to the regular expressionparser must also be escaped with a backslash. Note: You can copy the code and run it in tclsh or wish to see the effects.

Example

```
#
# Examine an overview of UNIX/Linux disks
#
set list1 [list \
{/dev/wd0a 17086 10958 5272 68% /}\
{/dev/wd0f 179824 127798 48428 73% /news}\
{/dev/wd0h 1249244 967818 218962 82% /usr}\
{/dev/wd0g 98190 32836 60444 35% /var}]

foreach line $list1 {
    regexp {[^ ]* *([0-9]+)[^/]*(/[a-z]*)} $line match size mounted
    puts "$mounted is $size blocks"
}
#
# Extracting a hexadecimal value ...
#
set line {Interrupt Vector? [32(0x20)]}
regexp "\[^\t]+\t\\\[\[0-9]+\\(0x(\[0-9a-fA-F]+)\\)]" $line match hexval
puts "Hex Default is: 0x$hexval"

#
# Matching the special characters as if they were ordinary
#
set str2 "abc^def"
regexp "\[^a-f]*def" $str2 match
puts "using \[^a-f] the match is: $match"
regexp "\[a-f^]*def" $str2 match
puts "using \[a-f^] the match is: $match"
regsub {\^} $str2 " is followed by: " str3
puts "$str2 with the ^ substituted is: \"$str3\""
```

```
  regsub "(\[a-f]+)\\^(\[a-f]+)" $str2 "\\2 follows \\1" str3
  puts "$str2 is converted to \"$str3\""
```

Result:

```
/ is 17086 blocks
/news is 179824 blocks
/usr is 1249244 blocks
/var is 98190 blocks
Hex Default is: 0x20
using [^a-f] the match is: ^def
using [a-f^] the match is: abc^def
abc^def with the ^ substituted is: "abc is followed by: def"
abc^def is converted to "def follows abc"
```

3.12 Associative Arrays.

Languages like C, BASIC, Fortran and Java support arrays in which the index value is an integer. Tcl, like most scripting languages (Perl, Python, PHP, etc...) supports associative arrays (also known as ”hash tables”) in which the index value is a string.

The syntax for an associative array is to put the index within parentheses:

```
set name(first) "Mary"
set name(last) "Poppins"

puts "Full name: $name(first) $name(last)"
```

There are several array commands aside from simply accessing and creating arrays which will be discussed in this and the next lesson.

‘array exists arrayName‘
Returns 1 if `arrayName` is an array variable. Returns 0 if `arrayName` is a scalar variable, proc, or does not exist.

‘array names arrayName ?pattern‘
Returns a list of the indices for the associative array `arrayName`. If `pattern` is supplied, only those indices that match `pattern` are returned. The match is done using the globbing technique from `string match`.

‘array size arrayName‘
Returns the number of elements in array `arrayName`.

‘array get arrayName‘
Returns a list in which each odd member of the list (1, 3, 5, etc) is an index into the associative array. The list element following a name is the value of that array member.

‘array set arrayName dataList‘
Converts a list into an associative array. `DataList` is a list in the format of that

returned by `array get`. Each odd member of the list (1, 3, 5, etc) is an index into the associative array, and the list element following that is the value of that array member.

‘array unset arrayName ?pattern¿
Unsets all of the elements in the array. If `pattern` exists, only the elements that match pattern are unset.

When an associative array name is given as the argument to the `global` command, all the elements of the associative array become available to that proc. For this reason, *Brent Welch* recommends (in ¡I¿Practical Programming in Tcl and Tk¡/I¿) using an associative array for the state structure in a package.

This method makes it simpler to share data between many procs that are working together, and doesn't pollute the global namespace as badly as using separate globals for all shared data items.

Another common use for arrays is to store tables of data. In the example below we use an array to store a simple database of names.

Examples

Example 1

```
proc addname {first last} {
    global name

    # Create a new ID (stored in the name array too for easy access)

    incr name(ID)
    set id $name(ID)

    set name($id,first) $first ;# The index is simply a string!
    set name($id,last) $last ;# So we can use both fixed and
                              ;# varying parts
}

#
# Initialise the array and add a few names
#
global name
set name(ID) 0

addname Mary Poppins
addname Uriah Heep
addname Rene Descartes
addname Leonardo "da Vinci"

#
# Check the contents of our database
# The parray command is a quick way to
# print it
#
```

```
parray name
```

Result:

```
name(1,first) = Mary
name(1,last) = Poppins
name(2,first) = Uriah
name(2,last) = Heep
name(3,first) = Rene
name(3,last) = Descartes
name(4,first) = Leonardo
name(4,last) = da Vinci
name(ID) = 4
```

Example 2

```
#
# Some array commands
#
array set array1 [list {123} {Abigail Aardvark} \
                     {234} {Bob Baboon} \
                     {345} {Cathy Coyote} \
                     {456} {Daniel Dog} ]

puts "Array1 has [array size array1] entries\n"

puts "Array1 has the following entries: \n [array names array1] \n"

puts "ID Number 123 belongs to $array1(123)\n"

if {[array exist array1]} {
   puts "array1 is an array"
} else {
   puts "array1 is not an array"
}

if {[array exist array2]} {
   puts "array2 is an array"
} else {
   puts "array2 is not an array"
}

proc existence {variable} {
   upvar $variable testVar
   if { [info exists testVar] } {
      puts "$variable Exists"
   } else {
      puts "$variable Does Not Exist"
   }
}
```

Result:

```
Array1 has 4 entries

Array1 has the following entries:
 345 234 123 456

ID Number 123 belongs to Abigail Aardvark

array1 is an array
array2 is not an array
```

Example 3

```
# Create an array
for {set i 0} {$i < 5} {incr i} { set a($i) test }

puts "\ntesting unsetting a member of an array"
existence a(0)
puts "a0 has been unset"
unset a(0)
existence a(0)

puts "\ntesting unsetting several members of an array, with an error"
existence a(3)
existence a(4)
catch {unset a(3) a(0) a(4)}
puts "\nAfter attempting to delete a(3), a(0) and a(4)"
existence a(3)
existence a(4)

puts "\nUnset all the array's elements"
existence a
array unset a *

puts "\ntesting unsetting an array"
existence a
puts "a has been unset"
unset a
existence a
```

Result:

```

testing unsetting a member of an array
a(0) Exists
a0 has been unset
a(0) Does Not Exist
```

```
testing unsetting several members of an array, with an error
a(3) Exists
a(4) Exists

After attempting to delete a(3), a(0) and a(4)
a(3) Does Not Exist
a(4) Exists

Unset all the array's elements
a Exists

testing unsetting an array
a Exists
a has been unset
a Does Not Exist
```

3.13 More Array Commands - Iterating and use in procedures

Often you will want to loop through the contents of an associative array - without having to specify the elements explicitly. For this the `array names` and `array get` commands are very useful. With both you can give a (glob-style) pattern to select what elements you need:

```
foreach name [array names mydata] {
    puts "Data on \"$name\": $mydata($name)"
}

#
# Get names and values directly
#
foreach {name value} [array get mydata] {
    puts "Data on \"$name\": $value"
}
```

Note, however, that the elements will not be returned in any predictable order: this has to do with the underlying "hash table". If you want a particular ordering (alphabetical for instance), use code like:

```
foreach name [lsort [array names mydata]] {
    puts "Data on \"$name\": $mydata($name)"
}
```

While arrays are great as a storage facility for some purposes, they are a bit tricky when you pass them to a procedure: they are actually collections of variables. This will not work:

```
proc print12 {a} {
   puts "$a(1), $a(2)"
}

set array(1) "A"
set array(2) "B"

print12 $array
```

Result:

```
can't read "array": variable is array
    while executing
"print12 $array"
    (file "xx.tcl" line 8)
```

The reason is very simple: an array does not have a value. Instead the above code should be:

```
proc print12 {array} {
   upvar $array a
   puts "$a(1), $a(2)"
}

set array(1) "A"
set array(2) "B"

print12 array
```

So, instead of passing a ”value” for the array, you pass the *name*. This gets aliased (via the upvar command) to a local variable (that behaves the as original array). You can make changes to the original array in this way too.

Example

```
#
# The example of the previous lesson revisited - to get a
# more general "database"
#

proc addname {db first last} {
    upvar $db name

    # Create a new ID (stored in the name array too for easy access)

    incr name(ID)
    set id $name(ID)

    set name($id,first) $first ;# The index is simply a string!
    set name($id,last) $last ;# So we can use both fixed and
                              ;# varying parts
}
```

```
proc report {db} {
    upvar $db name

    # Loop over the last names: make a map from last name to ID

    foreach n [array names name "*,last"] {
        #
        # Split the name to get the ID - the first part of the name!
        #
        regexp {^[^,]+} $n id

        #
        # Store in a temporary array:
        # an "inverse" map of last name to ID)
        #
        set last $name($n)
        set tmp($last) $id
    }

    #
    # Now we can easily print the names in the order we want!
    #
    foreach last [lsort [array names tmp]] {
        set id $tmp($last)
        puts " $name($id,first) $name($id,last)"
    }
}

#
# Initialise the array and add a few names
#
set fictional_name(ID) 0
set historical_name(ID) 0

addname fictional_name Mary Poppins
addname fictional_name Uriah Heep
addname fictional_name Frodo Baggins

addname historical_name Rene Descartes
addname historical_name Richard Lionheart
addname historical_name Leonardo "da Vinci"
addname historical_name Charles Baudelaire
addname historical_name Julius Caesar

#
# Some simple reporting
#
puts "Fictional characters:"
report fictional_name
puts "Historical characters:"
```

```
report historical_name
```

Result:

```
Fictional characters:
   Frodo Baggins
   Uriah Heep
   Mary Poppins
Historical characters:
   Charles Baudelaire
   Julius Caesar
   Rene Descartes
   Richard Lionheart
   Leonardo da Vinci
```

3.14 Dictionaries as alternative to arrays

- They cannot be passed directly to a procedure as a value. Instead you have to use the `array get` and `array set` commands to convert them to a value and back again, or else use the `upvar` command to create an alias of the array.

- Multidimensional arrays (that is, arrays whose index consists of two or more parts) have to be emulated with constructions like:

```
set array(foo,2) 10
set array(bar,3) 11
```

The comma used here is not a special piece of syntax, but instead just part of the string key. In other words, we are using a one-dimensional array,with keys like ”foo,2” and ”bar,3”. This is quite possible, but it can become very clumsy (there can be no intervening spaces for instance).

- Arrays cannot be included in other data structures, such as lists, or sent over a communications channel, without first packing and unpacking them into a string value.

The alternative is the `dict` command. This provides efficient access to key-value pairs, just like arrays, but these *dictionaries* are pure values. This means that you can pass them to a procedure just as a list or a string, without the need for `dict`. Tcl dictionaries are therefore much more like Tcl lists, except that they represent a mapping from keys to values, rather than an ordered sequence.

Unlike arrays, you can nest dictionaries, so that the value for a particular key consists of another dictionary. That way you can elegantly build complicated data structures, such as hierarchical databases. You can also combine dictionaries with other Tcl data structures. For instance, you can build a list of dictionaries that themselves contain lists.

Here is an example (adapted from the man page):

```
#
# Create a dictionary:
# Two clients, known by their client number,
# with forenames, surname
#
dict set clients ID1 forenames Joe
dict set clients ID1 surname Schmoe
dict set clients ID2 forenames Anne
dict set clients ID2 surname Other

#
# Print a table
#
puts "Number of clients: [dict size $clients]"
dict for {id info} $clients {
    puts "Client $id:"
    dict with info {
       puts " Name: $forenames $surname"
    }
}
```

- What happens in this example is: We fill a dictionary, called *clients*, with the information we have on two clients. The dictionary has two keys, "1" and "2" and the value for each of these keys is itself a (nested) dictionary – again with two keys: "forenames" and "surname". The `dict set` command accepts a list of key names which act as a path through the dictionary. The last argument to the command is the value that we want to set. You can supply as many key arguments to the `dict set` command as you want, leading to arbitrarily complicated nested data structures. Be careful though! Flat data structure designs are usually better than nested for most problems.

- The `dict for` command then loops through each key and value pair in the dictionary (at the outer-most level only). `dict for` is essentially a version of `foreach` that is specialised for dictionaries. We could also have written this line as: `foreach {id info} \$clients { ... }`. This takes advantage of the fact that, in Tcl, every dictionary is also a valid Tcl list, consisting of a sequence of name and value pairs representing the contents of the dictionary. The `dict for` command is preferred when working with dictionaries, however, as it is both more efficient, and makes it clear to readers of the code that we are dealing with a dictionary and not just a list.

- To get at the actual values in the dictionary that is stored with the client IDs we use the `dict with` command. This command takes the dictionary and unpacks it into a set of local variables in the current scope. For instance, in our example, the "info" variable on each iteration of the outer loop will contain a dictionary with two keys: "forenames" and "surname". The `dict with` command unpacks these keys into local variables with the same name as the key and with the associated value from the dictionary. This allows us to use a more convenient syntax when accessing the values, instead of having to use `dict get` everywhere. A related command is the `dict update` command, that allows you to specify exactly which keys you want to convert into variables. Be aware that any changes you make to these variables will be copied back into the dictionary when the `dict with` command finishes.

The order in which elements of a dictionary are returned during a `dict for` loop is defined to be the chronological order in which keys were added to the dictionary. If you need to access the keys in some other order, then it is advisable to explicitly sort the keys first. For example, to retrieve all elements of a dictionary in alphabetical order, based on the key, we can use the `lsort` command:

```
foreach name [lsort [dict keys $mydata]] {
    puts "Data on \"$name\": [dict get $mydata $name]"
}
```

Example

In this example, we convert the simple database of the previous lessons to work with dictionaries instead of arrays.

```
#
# The example of the previous lesson revisited - using dicts.
#

proc addname {dbVar first last} {
    upvar 1 $dbVar db

    # Create a new ID (stored in the name array too for easy access)
    dict incr db ID
    set id [dict get $db ID]

    # Create the new record
    dict set db $id first $first
    dict set db $id last $last
}

proc report {db} {

    # Loop over the last names: make a map from last name to ID

    dict for {id name} $db {
        # Create a temporary dictionary mapping from
        # last name to ID, for reverse lookup
        if {$id eq "ID"} { continue }
        set last [dict get $name last]
        dict set tmp $last $id
    }

    #
    # Now we can easily print the names in the order we want!
    #
    foreach last [lsort [dict keys $tmp]] {
        set id [dict get $tmp $last]
        puts " [dict get $db $id first] $last"
    }
}

#
# Initialise the array and add a few names
#
dict set fictional_name ID 0
dict set historical_name ID 0

addname fictional_name Mary Poppins
addname fictional_name Uriah Heep
addname fictional_name Frodo Baggins

addname historical_name Rene Descartes
addname historical_name Richard Lionheart
addname historical_name Leonardo "da Vinci"
addname historical_name Charles Baudelaire
addname historical_name Julius Caesar

#
# Some simple reporting
#
puts "Fictional characters:"
report $fictional_name
puts "Historical characters:"
report $historical_name
```

Note that in this example we use dictionaries in two different ways. In the `addname` procedure, we pass the dictionary variable by name and use `upvar` to make a link to it, as we did previously for arrays. We do this so that changes to the database are reflected in the calling scope, without having to return a new dictionary value. (Try changing the code to avoid using `upvar`). In the `report` procedure, however, we pass the dictionary as a value and use it directly. Compare the dictionary and array versions of this example (from the *previous lesson* (Section 3.13)) to see the differences between the two data structures and how they are used.

Chapter 4

Input and Output

4.1 File Access 101

Tcl provides several methods to read from and write to files on disk. The simplest methods to access a file are via `gets` and `puts`. When there is a lot of data to be read, however, it is sometimes more efficient to use the `read` command to load an entire file, and then parse the file into lines with the `split` command.

These methods can also be used for communicating over sockets and pipes. It is even possible, via the so-called *virtual file system* to use files stored in memory rather than on disk. Tcl provides an almost uniform interface to these very different resources, so that in general you do not need to concern yourself with the details.

`open fileName ?access? ?permission?`
Opens a file and returns a token to be used when accessing the file via `gets`, `puts`, `close`, etc.

- `fileName` is the name of the file to open.
- `access` is the file access mode

%Mode	Meaning%
`r`	Open the file for reading. The file must already exist.
`r+`	Open the file for reading and writing. The file must already exist.
`w`	Open the file for writing. Create the file if it doesn't exist, or set the length to zero if it does exist.
`w+`	Open the file for reading and writing. Create the file if it doesn't exist, or set the length to zero if it d
`a`	Open the file for writing. The file must already exist. Set the current location to the end of the file.
`a+`	Open the file for writing. Create the file if it does not exist. Set the current location to the end of the

- `permission` is an integer to use to set the file access permissions (most useful under Linux and other UNIX-like systems). The default is `rw-rw-rw-` (0666). You can use it to set the permissions for the file's owner, the group he/she belongs to and for all the other users. For many applications, the default is fine.

`close fileID`
Closes a file previously opened with `open`, and flushes any remaining output.

`gets fileID ?varName?`
Reads a line of input from `fileID`, and discards the terminating newline. If there is a `varName` argument, `gets` returns the number of characters read (or -1 if an EOF occurs), and places the line of input in `varName`. If `varName` is not specified, `gets` returns the line of input. An empty string will be returned if:

- There is a blank line in the file.
- The current location is at the end of the file. (An EOF occurs.) A typical loop to read the file line-by-line:

```
set infile [open "myfile.txt"]
while { gets $infile line] >= 0 } {
    ...
}
```

`puts ?-nonewline? ?fileID? string`
Writes the characters in string to the stream referenced by `fileID`, where `fileID` is one of:

- The value returned by a previous call to `open` with write access.
- `stdout` (standard output, the screen mostly)
- `stderr` (standard error, also the screen)

`read ?-nonewline? fileID`
Reads all the remaining bytes from `fileID`, and returns that string. If `-nonewline` is set, then the last character will be discarded if it is a newline. Any existing end of file condition is cleared before the `read` command is executed.

`read fileID numBytes`
Reads up to `numBytes` from `fileID`, and returns the input as a Tcl string. Any existing end of file condition is cleared before the `read` command is executed.

`seek fileID offset ?origin?`
Change the current position within the file referenced by `fileID`. Note that if the file was opened with "a" `access` that the current position can not be set before the end of the file for writing, but can be set to the beginning of the file for reading.

- `fileID` is one of:

** a file identifier returned by `open` ** `stdin` (standard input, the keyboard mostly) ** `stdout` ** `stderr`

- `offset` is the offset in bytes at which the current position is to be set. The position from which the offset is measured defaults to the start ofthe file, but can be from the current location, or the end by setting `origin` appropriately.
- `origin` is the position to measure `offset` from. It defaults to the start of the file. `origin` must be one of:

** `start` - the `offset` is measured from the start of the file. ** `current` - the `offset` is measured from the current position in the file. ** `end` - the `offset` is measured from the end of the file.

`tell fileID`
Returns the position of the access pointer in `fileID`.

`flush fileID`
Flushes any output that has been buffered for `fileID`.

`eof` **fileID‘**
Returns 1 if an End Of File condition exists, otherwise returns 0.
Points to remember about Tcl file access:

- The file I/O is buffered. The output may not be sent out when you expect it to be sent. Files will all be closed and flushed when your program exits normally, but may only be closed (not flushed) if the program is terminated in an unexpected manner.
- There are a finite number of open file slots available. While the exact number depends on the operating system you are using and its configuration, in practice you can have several hundreds of open files at a time. If you need more than that, you may need to reexamine your program.
- An empty line is indistinguishable from an EOF with the command: `set string [gets filename]` Use the `eof` command todetermine if the file is at the end or use the other form of `gets` (see the example).
- You can’t overwrite any data in a file that was opened with‘a‘ (append) `access`. You can, however, seek to the beginning of the file for `gets` commands.
- Opening a file with the `w+` access will allow you to overwrite data, but will delete all existing data in the file.
- Opening a file with the `r+` access will allow you to overwrite data, while saving the existing data in the file.
- By default the commands assume that strings represent ”readable” text. If you want to read ”binary” data, you will have to use the `fconfigure` command.
- Often, especially if you deal with configuration data for your programs, you can use the `source` command instead of the relatively low-level commands presentedhere. Just make sure your data can be interpreted as Tcl commands and ”source” the file.

Example

```
#
# Count the number of lines in a text file
#
set infile [open "myfile.txt" r]
set number 0

#
# gets with two arguments returns the length of the line,
# -1 if the end of the file is found
#
while { [gets $infile line] >= 0 } {
    incr number
}
close $infile

puts "Number of lines: $number"

#
# Also report it in an external file
#
set outfile [open "report.out" w]
puts $outfile "Number of lines: $number"
close $outfile
```

4.2 Information about Files - file, glob

There are two commands that provide information about the file system, `glob` and `file`. They provide a method to examine and manipulate the files and directories that is virtually indepedent of the operating system. Some subcommands and pieces of information are, however, necessarily system-dependent, as they refer to features of a particular operating system.

`glob` returns the names of files and subdirectories in a directory. It uses a name matching mechanism similar to the UNIX `ls` command or the Windows (DOS) `dir` command, to return a list of names that match a pattern.

`file` provides three sets of functionality:

- String manipulation appropriate to parsing file names

- Information about an entry in a directory:

- Manipulating the files and directories themselves:

Between these two commands, a program can obtain most of the information that it needs and can manipulate the files and directories. While retrieving information about what files are present and what properties they have is usually a highly platform-dependent matter, Tcl provides an interface that hides

Subcommand	Purpose
`dirname`	Returns directory portion of path
`extension`	Returns file name extension
`join`	Join directories and the file name to one string
`nativename`	Returns the *native* name of the file/directory
`rootname`	Returns file name without extension
`split`	Split the string into directory and file names
`tail`	Returns filename without directory

Subcommand	Purpose
`atime`	Returns time of last access
`executable`	Returns 1 if file is executable by the user
`exists`	Returns 1 if file exists, 0 otherwise
`isdirectory`	Returns 1 if entry is a directory, 0 otherwise
`isfile`	Returns 1 if entry is a regular file, 0 if not
`lstat`	Returns array of file status information
`mtime`	Returns time of last data modification
`owned`	Returns 1 if file is owned by the user
`readable`	Returns 1 if file is readable by the user
`readlink`	Returns name of file pointed to by a symbolic link
`size`	Returns file size in bytes
`stat`	Returns array of file status information
`type`	Returns type of file
`writable`	Returns 1 if file is writeable by the user

Subcommand	Purpose
`copy`	Copy a file or a directory
`delete`	Delete a file or a directory
`mkdir`	Create a new directory
`rename`	Rename or move a file or directory

almost all details that are specific to the platform (but are irrelevant to the programmer).

To take advantage of this feature, always manipulate file names via the `file join` and `file split` commands and the others in the first category.

For instance to refer to a file in a directory upwards of the current one:

```
set upfile [file join ".." "myfile.out"]
# upfile will have the value "../myfile.out"
```

(The ”..” indicates the ”parent directory”) Because external commands may not always deal gracefully with the uniform representation that Tcl employs (with forward slashes as directory separators), Tcl also provides a command to turn the string into one that is native to the platform:

```
#
# On Windows the name becomes "..\myfile.out"
#
set newname [file nativename [file join ".." "myfile.out"]]
```

Retrieving all the files with extension ”.tcl” in the current directory:

```
set tclfiles [glob *.tcl]
puts "Name - date of last modification"
foreach f $tclfiles {
    puts "$f - [clock format [file mtime $f] -format %x]"
}
```

(The clock command turns the number of seconds returned by the `file mtime` command into a simple date string, like "07/01/2017")

`glob ?switches? pattern ?pattern2 ...?`
Returns a list of file names that match `pattern` or `pattern2`, ... The `switches` may be one of the following (there are more switches available):

- `-nocomplain` allows `glob` to return an empty list without causing an error. Without this flag, an error would be generated when the empty list was returned.
- `-types typeList` selects which type of files/directory the command should return. The `typeList` may consist of type letters, like a "d" for directories and "f" for ordinary files as well as letters and keywords indicating the user's permissions ("r" for files/directories that can be read for instance).
- `--` marks the end of switches. This allows the use of "-" in a pattern without confusing the glob parser.

The `pattern` follows the same matching rules as the string match globbing rules with these exceptions:

- `{a,b,...}` Matches any of the strings a,b, etc.
- A "." at the beginning of a filename must match a "." in the filename. The "." is only a wildcard if it is not the first character in a name.
- All "/" must match exactly.
- If the first two characters in `pattern` are `~/`, then the `~` is replaced by the value of the `HOME` environment variable.
- If the first character in `pattern` is a' ', followed by a login id, then the `~loginid` is replaced by the path of loginid's home directory.

Note that the filenames that match `pattern` are returned in an arbitrary order (that is, do not expect them to be sorted in alphabetical order, for instance).

`file atime name`
Returns the number of seconds since some system-dependent start date, also known as the "epoch" (frequently 1/1/1970) when the file `name` was last accessed. Generates an error if the file doesn't exist, or the access time cannot be queried.

`file copy ?-force? name target`
Copy the file/directory `name` to a new file `target` (or to an existing directory with that name). The switch `-force` allows you to overwrite existing files.

`file delete ?-force? name`
Delete the file/directory `name`. The switch `-force` allows you to delete non-empty directories.

`file dirname name`
Returns the directory portion of a path/filename string. If `name` contains no slashes, `file dirname` returns a ".".

`file executable name`
Returns 1 if file `name` is executable by the current user, otherwise returns 0.

`file exists name`
Returns 1 if the file `name` exists, and the user has search access in all the directories leading to the file. Otherwise, 0 is returned.

`file extension name`
Returns the file extension.

`file isdirectory name`
Returns 1 if file name is a directory, otherwise returns 0.

`file isfile name`
Returns 1 if file name is a regular file, otherwise returns 0.

`file lstat name varName`
This returns the same information returned by the system call `lstat`. The results are placed in the associative array `varName`. The indexes in `varName` are:

- `atime` - time of last access
- `ctime` - time of last file status change
- `dev` - inode's device
- `gid` - group ID of the file's group
- `ino` - inode's number
- `mode` - inode protection mode
- `mtime` - time of last data modification
- `nlink` - number of hard links
- `size` - file size, in bytes
- `type` - type of file
- `uid` - user ID of the file's owner

Note: Because this calls `lstat`, if `name` is a symbolic link,the values in `varName` will refer to the link, not the file that is linked to.(See also the `stat` subcommand)

`file mkdir name`
Create a new directory `name`.

`file mtime name`
Returns the time of the last modification in seconds since Jan 1, 1970 or whatever start date (also known as epoch) the system uses.

`file owned name`
Returns 1 if the file is owned by the current user, otherwise returns 0.

`file readable name`
Returns 1 if the file is readable by the current user, otherwise returns 0.

`file readlink name`
Returns the name of the file a symlink is pointing to. If `name` isn't a symlink, or can't be read, an error is generated.

`file rename \verb`**?-force? name target'**
Rename file/directory `name` to the new name `target` (or to an existing directory with that name). The switch `-force` allows you to overwrite existing files.

`file rootname name`
Returns all the characters in `name` up to but not including the last ".". Returns the name if `name` doesn't include a ".".

`file size name`
Returns the size of `name` in bytes.

`file stat name varName`
This returns the same information returned by the system call `stat`. The results are placed in the associative array `varName`. The indexes in `varName` are:

- `atime` - time of last access
- `ctime` - time of last file status change
- `dev` - inode's device
- `gid` - group ID of the file's group
- `ino` - inode's number
- `mode` - inode protection mode
- `mtime` - time of last data modification
- `nlink` - number of hard links
- `size` - file size in bytes

- `type` - type of file
- `uid` - user ID of the file's owner

`file tail name`
Returns all of the characters in `name` after the last slash. Returns the name if `name` contains no slashes.

`file type name`
Returns a string giving the type of file name, which will be one of:

- `file` - normal file
- `directory` - directory
- `characterSpecial` - character oriented device
- `blockSpecial` - block oriented device
- `fifo` - named pipe
- `link` - symbolic link
- `socket` - named socket

`file writable name`
Returns 1 if file name is writable by the current user, otherwise returns 0.

Note: The overview given above does not cover all the details of the various subcommands, nor does it list all subcommands. Please check the man pages for these.

Example

```
#
# Report all the files and subdirectories in the current directory
# For files: show the size
# For directories: show that they _are_ directories
#
set dirs [glob -nocomplain -type d *]
if { [llength $dirs] > 0 } {
    puts "Directories:"
    foreach d [lsort $dirs] {
        puts " $d"
    }
} else {
    puts "(no subdirectories)"
}
set files [glob -nocomplain -type f *]
if { [llength $files] > 0 } {
    puts "Files:"
    foreach f [lsort $files] {
        puts " [file size $f] - $f (extension: [file extension $f])"
    }
} else {
    puts "(no files)"
}
```

Result: The output of the above example for an arbitrary directory:

```
Directories:
    CVS
    figs
    html
    latex
    pdf
Files:
    36 - .cvsignore (extension: .cvsignore)
    4 - .tex2page.hdir (extension: .hdir)
    461 - Makefile (extension: )
    509 - README.txt (extension: .txt)
    2708 - refs.bib (extension: .bib)
```

4.3 Running other programs from Tcl - exec, open

So far the lessons have dealt with programming within the Tcl interpreter. However, Tcl is also useful as a scripting language to tie other packages or programs together. To accomplish this function, Tcl has two ways to start another program:

- `open` - run a new program and open a file-like connection to this program.
- `exec` - run a new program as a more or less independent subprocess

The `open` call is the same call that is used to open a file. If the first character in the file name argument is a ”pipe” symbol (`|`), then `open` will treat the rest of the argument as a program name, and will run that program with the *standard input* or *output* connected to a file descriptor. This ”pipe” connection can be used to read the output from that other program or to write fresh input data to it or both.

If the ”pipe” is opened for both reading and writing you must be aware that the pipes are buffered. The output from a `puts` command will be saved in an I/O buffer until the buffer is full, or until you execute a `flush` command to force it to be transmitted to the other program. The output of this other program will not be available to a `read` or `gets` until its output buffer is filled up or flushed explicitly.

(*Note:* as this is *internal* to this other program, there is no way that your Tcl script can influence that. The other program simply must cooperate. Well, that is not entirely true: the *Expect* extension actually works around this limitation by exploiting deep system features.)

The `exec` call is similar to invoking a program (or a set of programs piped together) from the prompt in an interactive shell or a DOS-box or in a UNIX/Linux shell script. It supports several styles ofoutput redirection, or it can return the output of the other program(s)as the return value of the `exec` call.

`open |program ?access?`
Returns a file descriptor for the pipe. The `program` argument must start with the pipe symbol. If `program` is enclosed in quotes or braces, it can include arguments to the subprocess.

`exec ?switches? arg1 ?arg2? ... ?argN?`
`exec` treats its arguments as the names and arguments for a set of programs to run. If the first `args` start with a `"-"`, then they are treated as `switches` to the `exec` command, instead of being invoked as subprocesses or subprocess options. `switches` are:

- `-keepnewline` retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.
- `--` marks the end of the switches. The next string will be treated as `arg1`, even if it starts with a "'-'"

The arguments to the `exec` command, `arg1` ... `argN` can be one of:

- the name of a program to execute
- a command line argument for the subprocess
- an I/O redirection instruction.
- an instruction to put the new program in the background:

```
exec myprog &
```

will start the program `myprog` in the background, and return immediately. There is no connection between that program and the Tcl script, both can run on independently. The `&` must be the last argument - you can use all other types of arguments in front of it.

[NOTE: Add information on how to wait for the program to finish?]

There are many I/O redirection commands. The main subset of these commands is: `|`

Pipes the *standard output* of the command preceding the pipe symbol into the *standard input* of the command following the pipe symbol.

`< filename`
The first program in the pipe will read input from `filename`.

`<\ fileID`
The first program in the pipe will read input from the Tcl descriptor `fileID`. `fileID` is the value returned from an `open ... "r"` command.

`<< value`
The first program in the pipe will read `value` as its input.

`> filename`
The output of the last program in the pipe will be sent to `filename`. Any previous contents of `filename` will be lost.

`>> filename`
The output of the last program in the pipe will be appended to `filename`.

`2> filename`
The *standard error* from all the programs in the pipe will be sent to `filename`. Any previous contents of `filename` will be lost.

`2>> filename`
The *standard error* from all the programs in the pipe will be appended to `filename`.

`>\ fileID`
The output from the last program in the pipe will be written to `fileID`. `fileID` is the value returned from an `open ... "w"` command.

If you are familiar with shell programming, there are a few differences to be aware of when you are writing Tcl scripts that use the `exec` and `open` calls.

- You don't need the quotes that you would put around arguments to escape them from the shell expanding them. In the example, the argument to the `sed` command is not put in quotes. If it were put in quotes, the quotes would be passed to `sed`, instead of being stripped off (as theshell does), and `sed` would report an error.
- If you use the `open |cmd "r+"` construct, you must follow each `puts` with a flush to force Tcl to send the command from its buffer to the program. The output from the program itself may be buffered in its output buffer. You can sometimes force the output from the external program to flush by sending an `exit` command to the process. You can also use the `fconfigure` command to make a connection (channel) unbuffered.
- This will fail - there is most probably no file with the literal name "*.tcl":

```
exec ls *.tcl
```

- To pass a list of files, based on such a pattern use the `{*}` prefix, it forces the list to become individual arguments:

```
exec ls {*}[glob *.tcl]
```

- If one of the commands in an `exec` call fails to execute, the `exec` will return an error, and the error output will include the last line describing the error. The `exec` treats any output to *standard error* to be an indication that the external program failed. This is simply a conservative assumption: many programs behave that way and they are sloppy in setting return codes. Some programs however write to *standard error* without intending this as an indication of an error. You can guard against this from upsetting your script by using the `catch`command:

```
if { [catch { exec ls *.tcl } msg] } {
    puts "Something seems to have gone wrong but we will ignore it"
}
```

As already mentioned, the *Expect* extension to Tcl provides a very powerful interface to other programs, which in particular handles the buffering problem. *NOTE:* add good reference to expect.

If one of the commands in an `open |cmd` fails, the `open` does not return an error. However, attempting to read input from the file descriptor with `gets \$file` will return an empty string. Using the `gets \$file input` construct will return a character count of -1.

To inspect the return code from a program and the possible reason for failure, you can use the global `errorInfo` variable:

```
if { [catch { exec ls *.tcl } msg] } {
    puts "Something seems to have gone wrong:"
    puts "Information about it: $::errorInfo"
}
```

Example

```
#
# Write a Tcl script to get a platform-independent program:
#
# Create a unique (mostly) file name for a Tcl program
set TMPDIR "."
if { [info exists ::env(TMP)] } {
     set TMPDIR $::env(TMP)
}
set tempFileName [file join $TMPDIR invert_[pid].tcl]

# Open the output file, and # write the program to it
set outfl [open $tempFileName w]
puts $outfl {
    set len [gets stdin line]
    if {$len < 5} {exit -1}
    for {set i [expr {$len-1}]} {$i >= 0} {incr i -1} {
        append invertedLine [string range $line $i $i]
    }
    puts $invertedLine
    exit 0
}
```

```
# Close the file
close $outfl

#
# Run the new Tcl script:
#
# Open a pipe to the program (for both reading and writing: r+)
#

set io [open "|[info nameofexecutable] $tempFileName" r+]

#
# Send a string to the new program
# *MUST FLUSH*

puts $io "This will come back backwards."
flush $io

# Get the reply, and display it.

set len [gets $io line]
puts "To invert: 'This will come back backwards.'"
puts "Inverted is: $line"
puts "The line is $len characters long"

# Run the program with input defined in an exec call

set invert [exec [info nameofexecutable] $tempFileName << \
       "ABLE WAS I ERE I SAW ELBA"]

# Display the results
puts "The inversion of 'ABLE WAS I ERE I SAW ELBA' is \n $invert"

# Clean up
file delete $tempFileName
```

Result:

```
To invert: 'This will come back backwards.'
Inverted is: .sdrawkcab kcab emoc lliw sihT
The line is 30 characters long
The inversion of 'ABLE WAS I ERE I SAW ELBA' is
 ABLE WAS I ERE I SAW ELBA
```

4.4 Communicating with other programs - socket, fileevent

TODO: lesson on socket and fileevent (or "chan event" in 8.5). But see also *Lesson 40: socket, fileevent, vwait* (Section 7.9)

Example

```
#
# A little echo server
#
proc accept {socket address port} {
   puts "Accepted connection $socket from $address\:$port"

   # Copy input from the socket directly back to the socket

   chan copy $socket $socket -command [list finish $socket]
}
proc finish {socket args} {
   puts "Closed $socket"
   catch { chan close $socket }
}
socket -server accept 8080
# Start the event loop by waiting on a non-existant variable 'forever'
vwait forever
```

Chapter 5

Input and Output

5.1 Learning the existence of commands and variables - info

Tcl provides a number of commands for *introspection* - commands that tell what is going on in your program, what the implementation is of your procedures, which variables have been set and so on.

The `info` command allows a Tcl program to obtain information from the Tcl interpreter. The next three lessons cover aspects of the `info` command. (Other commands allowing introspection involve: traces, namespaces, commands scheduled for later execution via the `after` command and so on.)

This lesson covers the `info` subcommands that return information about which procs, variables, or commands are currently in existence in this instance of the interpreter. By using these subcommands you can determine if a variable or proc exists before you try to access it.

The code below shows how to use the `info exists` command to make an `incr` command that throws a *no such variable* error, since it checks to be certain that the variable exists before incrementing it (the core command initialises the variable to zero instead):

```
proc nitpickyIncr {val {amount 1}} {
    upvar $val v
    if { [info exists v] } {
        incr v $amount
    } else {
        return -code error "Variable $val does not exist!"
    }
}
```

Info commands that return lists of visible commands and variables.

Almost all the `info` subcommands take a pattern that follow the `string match` rules. If `pattern` is not provided, a list of all items is returned (as if the pattern was "*").

`info commands ?pattern?`
Returns a list of the commands, both internal commands and procedures, whose names match `pattern`.

`info exists varName`
Returns 1 if `varName` exists as a variable (or an array element) in the current context, otherwise returns 0.

`info functions ?pattern?`
Returns a list of the mathematical functions available via the `expr` command that match `pattern`.

`info globals ?pattern?`
Returns a list of the global variables that match `pattern`.

`info locals ?pattern?`
Returns a list of the local variables that match `pattern`.

`info procs ?pattern?`
Returns a list of the Tcl procedures that match `pattern`.

`info vars  ?pattern?`
eturns a list of the local and global variables that match `pattern`.

Example

```
#
# Use the catch command to prevent the program from ending prematurely
#
if {[info procs nitpickyIncr] eq "nitpickyIncr"} {
    catch {
        nitpickyIncr a
    } msg
}

puts "After calling nitpickyIncr with a non existent variable: $msg"

set a 100
nitpickyIncr a
puts "After calling nitpickyIncr with a variable with a value of 100: $a"

catch {
    nitpickyIncr b -3
} msg
puts "After calling nitpickyIncr with a non existent variable by -3: $msg"

set b 100
nitpickyIncr b -3
puts "After calling nitpickyIncr with a variable whose value is 100 by -3: $b"
puts "\nThese variables have been defined: [lsort [info vars]]"
puts "\nThese globals have been defined: [lsort [info globals]]"

set exist [info procs localproc]
if {$exist == ""} {
    puts "\nlocalproc does not exist at point 1"
}

proc localproc {} {
    global argv

    set loc1 1
    set loc2 2
    puts "\nLocal variables accessible in this proc are: [lsort [info locals]]"
    puts "\nVariables accessible from this proc are: [lsort [info vars]]"
    puts "\nGlobal variables visible from this proc are: [lsort [info globals]]"
}

set exist [info procs localproc]
if {$exist != ""} {
    puts "localproc does exist at point 2"
}

localproc
```

Result:

(Note: Some lines have been abbreviated to make them fit.)

```
After calling nitpickyIncr with a non existent variable: Variable a does not exist!
After calling nitpickyIncr with a variable with a value of 100: 101
After calling nitpickyIncr with a non existent variable by -3: Variable b does not exist!
After calling nitpickyIncr with a variable whose value is 100 by -3: 97

These variables have been defined: a argc argv argv0 auto_path b env errorCode ...

These globals have been defined: a argc argv argv0 auto_path b env errorCode ...

localproc does not exist at point 1
localproc does exist at point 2

Local variables accessible in this proc are: loc1 loc2

Variables accessible from this proc are: argv loc1 loc2

Global variables visible from this proc are: a argc argv argv0 auto_path b env errorCode ...
```

5.2 State of the interpreter - info

There are a number of subcommands to the `info` command that provide information about the current state of the interpreter. These commands provide access to information like the current version and patchlevel, what script is currently being executed, how many commands have been executed, or how far down in the call tree the current proc is executing. The `info tclversion` and `info patchlevel` can be used to find out if the revision level of the interpreter running your code has the support for features you are using. If you know that certain features are not available in certain revisions of the interpreter, you can define your own procs to handle this, or just exit the program with an error message.

The `info cmdcount` and `info level` can be used while optimizing a Tcl script to find out how many levels and commands were necessary to accomplish a function.

Note that the `pid` command is not part of the `info` command, but a command in its own right.

Subcommands that return information about the current state of the interpreter

(Note: There are several other subcommands that can be useful at times)

`info cmdcount`
Returns the total number of commands that have been executed by this interpreter.

`info level ?number?`
Returns the stack level at which the compiler is currently evaluating code. 0 is the top level, 1 is a proc called from top, 2 is a proc called from a proc, etc. If `number` is a positive value, `info level` returns a the name and arguments of the proc at that level on the stack. `number` is that same value that `info level` would return if it were called in the proc being referenced. If `number` number is a negative value, it refers to the current level plus `number`. Thus, `info level` returns a the name and arguments of the proc at that level on the stack.

`info patchlevel`
Returns the value of the global variable tcl$_patchlevel.Thisisathree-levelsversionnumberidentifyingth$
"8.6.6"

`info script`
Returns the name of the file currently being evaluated, if one is being evaluated. If there is no file being evaluated, returns an empty string. This can be used for instance to determine the directory holding other scripts or files of interest (they often live in the same directory or in a related directory), without having to hardcode the paths.

`info tclversion`
Returns the value of the global variable tcl$_version.Thisistherevisionnumberofthisinterpreter, like :$
"8.6".

`pid`
Returns the process id of the current process.

Example

```
puts "This is how many commands have been executed: [info cmdcount]"
puts "Now *THIS* many commands have been executed: [info cmdcount]"

puts "\nThis interpreter is revision level: [info tclversion]"
puts "This interpreter is at patch level: [info patchlevel]"

puts "The process id for this program is [pid]"

proc factorial {val} {
    puts "Current level: [info level] - val: $val"
    set lvl [info level]
    if {$lvl == $val} {
        return $val
    }
    return [expr {($val-$lvl) * [factorial $val]}]
}

set count1 [info cmdcount]
set fact [factorial 3]
set count2 [info cmdcount]
puts "The factorial of 3 is $fact"
puts "Before calling the factorial proc, $count1 commands had been executed"
puts "After calling the factorial proc, $count2 commands had been executed"
puts "It took [expr $count2-$count1] commands to calculate this factorial"
```

Result:

```
This is how many commands have been executed: 147
Now *THIS* many commands have been executed: 150

This interpreter is revision level: 8.6
This interpreter is at patch level: 8.6.6
The process id for this program is 10816
Current level: 1 - val: 3
Current level: 2 - val: 3
Current level: 3 - val: 3
The factorial of 3 is 6
Before calling the factorial proc, 162 commands had been executed
After calling the factorial proc, 189 commands had been executed
It took 27 commands to calculate this factorial
```

The `info script` command is useful to construct the names of related script files:

```
#
# Use [info script] to determine where the other files of interest
# reside
#
set sysdir [file dirname [info script]]
source [file join $sysdir "utils.tcl"]
```

5.3 Information about procs - info

The `info` command includes a set of subcommands that will provide all the info you could want about a proc. These subcommands will return the body of a proc, the arguments to the proc, and the value of any default arguments.

These subcommands can be used to:

- Access the contents of a proc in a debugger.
- Generate custom procs from a template.
- Report default values while prompting for input.

Info commands that return information about a proc

`info args procname`
Returns a list of the names of the arguments to the procedure `procname`.

`info body procname`
Returns the body of the procedure `procname`.

`info default procname arg varName`
Returns 1 if the argument `arg` in procedure `procName` has a default, and sets `varName` to that default. Otherwise, returns 0.

Example

```
proc demo {argument1 {default "DefaultValue"} } {
    puts "This is a demo proc. It is being called with $argument1 and $default"
    #
    # We can use [info level] to find out if a value was given for
    # the optional argument "default" ...
    #
    puts "Actual call: [info level [info level]]"
}

puts "The args for demo are: [info args demo]\n"
puts "The body for demo is: [info body demo]\n"

set arglist [info args demo]

foreach arg $arglist {
    if { [info default demo $arg defaultval] } {
       puts "$arg has a default value of $defaultval"
    } else {
       puts "$arg has no default"
    }
}
```

Result:

```
The args for demo are: argument1 default

The body for demo is:
    puts "This is a demo proc. It is being called with $argument1 and $default"
    #
    # We can use [info level] to find out if a value was given for
    # the optional argument "default" ...
    #
    puts "Actual call: [info level [info level]]"


argument1 has no default
default has a default value of DefaultValue
```

Chapter 6

Modularization - source

6.1 Modularization - source

The `source` command will load a file and execute it. This allows a program to be broken up into multiple files, with each file defining procedures and variables for a particular area of functionality. For instance, you might have a file called `database.tcl` that contains all the procedures for dealing with a database, or a file called `gui.tcl` that handles creating a graphical user interface with Tk. The main script can then simply include each file using the `source` command. More powerful techniques for program modularization are discussed in the next lesson on packages.

This command can be used to:

- separate a program into multiple files.
- make a library file that contains all the procs for a particular set of functions.
- configure programs.
- load data files.

`source fileName`
Reads the script in `fileName` and executes it.

- If the script executes successfully, `source` returns the value of the last statement in the script.
- If there is an error in the script, `source` will return that error.
- If there is a return (other than within a `proc` definition) then `source` will return immediately, without executing the remainder of the script.
- If `fileName` starts with a tilde ( ) then `\$env(HOME)` will be substituted for the tilde, as is done in the `file` command.

Example

sourcedata.tcl:

```
# Example data file to be sourced
set scr [info script]
proc testproc {} {
    global scr
    puts "testproc source file: $scr"
}
set abc 1
return
set aaaa 1
```

sourcemain.tcl:

```
set filename "sourcedata.tcl"
puts "Global variables visible before sourcing $filename:"
puts "[lsort [info globals]]\n"

if {[info procs testproc] eq ""} {
    puts "testproc does not exist. sourcing $filename"
    source $filename
}

puts "\nNow executing testproc"
testproc

puts "Global variables visible after sourcing $filename:"
puts "[lsort [info globals]]\n"
```

Result: (slightly edited to make the long list fit)

```
Global variables visible before sourcing sourcedata.tcl:
argc argv argv0 auto_path env errorCode errorInfo filename tcl_interactive
tcl_library tcl_patchLevel tcl_platform tcl_rcFileName tcl_version

testproc does not exist. sourcing sourcedata.tcl

Now executing testproc
testproc source file: sourcedata.tcl
Global variables visible after sourcing sourcedata.tcl:
abc argc argv argv0 auto_path env errorCode errorInfo filename scr
tcl_interactive tcl_library tcl_patchLevel tcl_platform tcl_rcFileName tcl_version
```

6.2 Building reusable libraries - packages and namespaces

The previous lesson showed how the `source` command can be used to separate a program into multiple files, each responsible for a different area of functionality.

This is a simple and useful technique for achieving modularity. However, there are a number of drawbacks to using the `source` command directly. Tcl provides a more powerful mechanism for handling reusable units of code called *packages*. A package is simply a bundle of files implementing some functionality, along with a *name* that identifies the package, and a *version number* that allows multiple versions of the same package to be present. A package can be a collection of Tcl scripts, or a binary library, or a combination of both.

Using packages

The `package` command provides the ability to use a package, compare package versions, and to register your own packages with an interpreter. A package is loaded by using the `package require` command and providing the package `name` and optionally a `version` number. The first time a script requires a package Tcl builds up a database of available packages and versions. It does this by searching for *package index* files in all of the directories listed in the `tcl_pkgPath` and `auto_path` global variables, as well as any subdirectories of those directories. Each package provides a file called `pkgIndex.tcl` that tells Tcl the names and versions of any packages in that directory, and how to load them if they are needed.

It is good style to start every script you create with a set of `package require` statements to load any packages required. This serves two purposes: making sure that any missing requirements are identified as soon as possible; and, clearly documenting the dependencies that your code has. Tcl and Tk are both made available as packages and it is a good idea to explicitly require them in your scripts even if they are already loaded as this makes your scripts more portable and documents the version requirements of your script.

Creating a package

There are three steps involved in creating a package:

- Adding a `package provide` statement to your script.
- Creating a `pkgIndex.tcl` file (note the capital-I, this is essential for OSes like Linux)
- Installing the package where it can be found by Tcl.

The first step is to add a `package provide` statement to your script. It is good style to place this statement at the top of your script. The `package provide` command tells Tcl the `name` of your package and the `version` being provided.

The next step is to create a `pkgIndex.tcl` file. This file tells Tcl how to load your package. In essence the index file is simply a Tcl file which is loaded into the interpreter when Tcl searches for packages. It should use the `package ifneeded` command register a script which will load the package when it is required. The `pkgIndex.tcl` file is evaluated globally in the interpreter when Tcl first searches for *any* package. For this reason it is very bad style for an index script to do anything other than tell Tcl how to load a package; index scripts should not define procs, require packages, or perform any other action which may affect the state of the interpreter.

The simplest way to create a `pkgIndex.tcl` script is to use the `pkg_mkIndex` command. The `pkg_mkIndex` command scans files which match a given `pattern` in a `directory` looking for `package provide` commands. From this information it generates an appropriate `pkgIndex.tcl` file in the directory.

Once a package index has been created, the next step is to move the package to somewhere that Tcl can find it. The `tcl_pkgPath` and `auto_path` global variables contain a list of directories that Tcl searches for packages. The package index and all the files that implement the package should be installed into a subdirectory of one of these directories. Alternatively, the `auto_path` variable can be extended at run-time to tell Tcl of new places to look for packages.

`package require ?-exact? name ?version?`
Loads the package identified by `name`. If the `-exact` switch is given along with a `version` number then only that exact package version will be accepted. If a `version` number is given, without the `-exact` switch then any version equal to or greater than that version (but with the same major version number) will be accepted. If no version is specified then the highest available version will be loaded. If a matching package can be found then it is loaded and the command returns the actual version number; otherwise it generates an error.

`package provide name ?version?`
If a `version` is given this command tells Tcl that this version of the package indicated by `name` is loaded. If a different version of the same package has already been loaded then an error is generated. If the `version` argument is omitted, then the command returns the version number that is currently loaded, or the empty string if the package has not been loaded.

`pkg_mkIndex ?-direct? ?-lazy? ?-load pkgPat? ?-verbose? dir ?pattern pattern ...?`
Creates a `pkgIndex.tcl` file for a package or set of packages. The command works by loading the files matching the `pattern`s in the directory, `dir` and seeing what new packages and commands appear. The command is able to handle both Tcl script files and binary libraries.

Namespaces

One problem that can occur when using packages, and particularly when using code written by others is that of *name collision.* This happens when two pieces of code try to define a procedure or variable with the same name. In Tcl when this occurs the old procedure or variable is simply overwritten. This is sometimes a useful feature, but more often it is the cause of bugs if the two definitions are not compatible. To solve this problem, Tcl provides a `namespace` command to allow commands and variables to be partitioned into separate areas, called *namespaces.* Each namespace can contain commands and variables which are local to that namespace and cannot be overwritten by commands or variables in other namespaces. When a command in a namespace is invoked it can see all the other commands and variables in its namespace, as well as those in the global namespace. Namespaces can also contain other namespaces. This allows a hierarchy of namespaces to be created in a similar way to a file system hierarchy, or the Tk widget hierarchy. Each namespace itself has a name which is visible in its parent namespace. Items in a namespace can be accessed by creating a path to the item. This is done by joining the names of the items with `::`. For instance, to access the variable `bar` in the namespace `foo`, you could use the path `foo::bar`. This kind of path is called a relative path because Tcl will try to follow the path *relative* to the current namespace. If that fails, and the path represents a command, then Tcl will also look relative to the global namespace. You can make a path *fully-qualified* by describing its exact position in the hierachy from the global namespace, which is named `::`. For instance, if our `foo` namespace was a child of the global namespace, then the fully-qualified name of `bar` would be `::foo::bar`. It is usually a good idea to use fully-qualified names when referring to any item outside of the current namespace to avoid surprises.

A namespace can *export* some or all of the command names it contains. These commands can then be *imported* into another namespace. This in effect creates a local command in the new namespace which when invoked calls the original command in the original namespace. This is a useful technique for creating short-cuts to frequently used commands from other namespaces. In general, a namespace should be careful about exporting commands with the same name as any built-in Tcl command or with a commonly used name.

Some of the most important commands to use when dealing with namespaces are:

`namespace eval path script`
This command evaluates the `script` in the namespace specified by `path`. If the namespace doesn't exist then it is created. The namespace becomes the current namespace while the script is executing, and any unqualified names will be resolved relative to that namespace. Returns the result of the last command in `script`.

`namespace delete ?namespace namespace ...?`
Deletes each namespace specified, along with all variables, commands and child namespaces it contains.

`namespace current`
Returns the fully qualified path of the current namespace.

`namespace export ?-clear? ?pattern pattern ...?`
Adds any commands matching one of the patterns to the list of commands exported by the current namespace. If the `-clear` switch is given then the export list is cleared before adding any new commands. If no arguments are given, returns the currently exported command names. Each pattern is a glob-style pattern such as `*`, `[a-z]*`, or `*foo*`.

`namespace import ?-force? ?pattern pattern ...?`
Imports all commands matching any of the patterns into the current namespace. Each pattern is a glob-style pattern such as `foo::*`, or `foo::bar`.

Using namespace with packages

William Duquette has an excellent guide to using namespaces and packages at
[”http://www.wjduquette.com/tcl/namespaces.html”¿http://www.wjduquette.com/tcl/namespaces.html].

In general, a package should provide a namespace as a child of the global namespace and put all of its commands and variables inside that namespace. A package shouldn't put commands or variables into the global namespace by default. It is also good style to give your package and the namespace it provides the same name, to avoid confusion.

Example

```
# Register the package
package provide tutstack 1.0
package require Tcl 8.5

# Create the namespace
namespace eval ::tutstack {
    # Export commands
    namespace export create destroy push pop peek empty

    # Set up state
    variable stack
    variable id 0
}

# Create a new stack
proc ::tutstack::create {} {
    variable stack
    variable id

    set token "stack[incr id]"
    set stack($token) [list]
    return $token
}

# Destroy a stack
proc ::tutstack::destroy {token} {
    variable stack

    unset stack($token)
}

# Push an element onto a stack
proc ::tutstack::push {token elem} {
    variable stack

    lappend stack($token) $elem
}

# Check if stack is empty
proc ::tutstack::empty {token} {
    variable stack
    set num [llength $stack($token)]
    return [expr {$num == 0}]
}

# See what is on top of the stack without removing it
proc ::tutstack::peek {token} {
    variable stack

    if {[empty $token]} {
        error "stack empty"
    }

    return [lindex $stack($token) end]
}

# Remove an element from the top of the stack
proc ::tutstack::pop {token} {
    variable stack

    set ret [peek $token]
    set stack($token) [lrange $stack($token) 0 end-1]
    return $ret
}
```

This example creates a package which provides a stack data structure.

```
package require tutstack 1.0

set stack [tutstack::create]
foreach num {1 2 3 4 5} { tutstack::push $stack $num }

while { ![tutstack::empty $stack] } {
    puts "[tutstack::pop $stack]"
}

tutstack::destroy $stack
```

And some code which uses it:

```
package require tutstack 1.0
package require Tcl 8.5

namespace eval ::tutstack {
    # Create the ensemble command
    namespace ensemble create
}

# Now we can use our stack through the ensemble command
set stack [tutstack create]
foreach num {1 2 3 4 5} { tutstack push $stack $num }
```

```
while { ![tutstack empty $stack] } {
    puts "[tutstack pop $stack]"
}

tutstack destroy $stack
```

Ensembles

A common way of structuring related commands is to group them together into a single command with sub-commands. This type of command is called an *ensemble* command, and there are many examples in the Tcl standard library. For instance, the `string` command is an ensemble whose sub-commands are `length`, `index`, `match` etc. Tcl 8.5 introduced a handy way of converting a namespace into an ensemble with the `namespace ensemble` command. This command is very flexible, with many options to specify exactly how sub-commands are mapped to commands within the namespace. The most basic usage is very straightforward, however, and simply creates an ensemble command with the same name as the namespace and with all exported procedures registered as sub-commands. To illustrate this, we will convert our stack data structure into an ensemble:

```
package require tutstack 1.0
package require Tcl 8.5

namespace eval ::tutstack {
    # Create the ensemble command
    namespace ensemble create
}

# Now we can use our stack through the ensemble command
set stack [tutstack create]
foreach num {1 2 3 4 5} { tutstack push $stack $num }

while { ![tutstack empty $stack] } {
    puts "[tutstack pop $stack]"
}

tutstack destroy $stack
```

As well as providing a nicer syntax for accessing functionality in a namespace, ensemble commands also help to clearly distinguish the public interface of a package from the private implementation details, as only exported commands will be registered as sub-commands and the ensemble will enforce this distinction. Readers who are familiar with object-oriented programming (OOP) will realise that the namespace and ensemble mechanisms provide many of the same encapsulation advantages. Indeed, in the past many OO extensions for Tcl built on top of the powerful namespace mechanism. The official object-oriented features in TclOO do something very similar.

Chapter 7

Further topics

7.1 Creating Commands - eval

One difference between Tcl and most other compilers is that Tcl will allow an executing program to create new commands and execute them while running.

A tcl command is defined as a list of strings in which the first string is a command or proc. Any string or list which meets this criteria can be evaluated and executed.

The `eval` command will evaluate a list of strings as though they were commands typed at the \% prompt or sourced from a file. The `eval` command normally returns the final value of the commands being evaluated. If the commands being evaluated throw an error (for example, if there is a syntax error in one of the strings), then `eval` will throw an error.

Note that either `concat` or `list` may be used to create the command string, but that these two commands will create slightly different command strings.

`eval arg1 ?arg2? ... ?argn?`
Evaluates `arg1` - `argn` as one or more Tcl commands. The `args` are concatenated into a string, and evaluated as a Tcl script. `eval` returns the result (or error code) of that evaluation.

Example

```
set cmd {puts "Evaluating a puts"}
puts "CMD IS: $cmd"
eval $cmd

if {[string match [info procs newProcA] ""] } {
    puts "\nDefining newProcA for this invocation"
    set num 0;
    set cmd "proc newProcA "
    set cmd [concat $cmd "{} {\n"]
    set cmd [concat $cmd "global num;\n"]
    set cmd [concat $cmd "incr num;\n"]
    set cmd [concat $cmd " return \"/tmp/TMP.[pid].\$num\";\n"]
    set cmd [concat $cmd "}"]
    eval $cmd
}

puts "\nThe body of newProcA is: \n[info body newProcA]\n"

puts "newProcA returns: [newProcA]"
puts "newProcA returns: [newProcA]"

#
# Define a proc using lists
#

if {[string match [info procs newProcB] ""] } {
    puts "\nDefining newProcB for this invocation"
    set cmd "proc newProcB "
    lappend cmd {}
    lappend cmd {global num; incr num; return $num;}

    eval $cmd
}

puts "\nThe body of newProcB is: \n[info body newProcB]\n"
puts "newProcB returns: [newProcB]"
```

Result:

```
CMD IS: puts "Evaluating a puts"
Evaluating a puts

Defining newProcA for this invocation

The body of newProcA is:
 global num; incr num; return "/tmp/TMP.10312.$num";

newProcA returns: /tmp/TMP.10312.1
newProcA returns: /tmp/TMP.10312.2

Defining newProcB for this invocation

The body of newProcB is:
global num; incr num; return $num;

newProcB returns: 3
```

7.2 More command construction - format, list

There may be some unexpected results when you try to compose command strings for `eval`.

For instance:

```
eval puts OK
```

would print the string OK. However,

```
eval puts Not OK
```

will generate an error.

The reason that the second command generates an error is that the `eval` uses `concat` to merge its arguments into a command string. This causes the two words `Not OK` to be treated as two arguments to `puts`. If there is more than one argument to `puts`, the first argument must be a file pointer.

Correct ways to write the second command include these:

```
eval [list puts {Not OK}]
eval [list puts "Not OK"]
set cmd "puts" ; lappend cmd {Not OK}; eval $cmd
```

As long as you keep track of how the arguments you present to `eval` will be grouped, you can use many methods of creating the strings for `eval`, including the `string` commands and `format`.

The recommended methods of constructing commands for `eval` is to use the `list` and `lappend` commands. These commands become difficult to use, however if you need to put braces in the command, as was done in the previous lesson.

The example from the previous lesson is re-implemented in the example code using lappend.

The completeness of a command can be checked with `info complete`. `Info complete` can also be used in an interactive program to determine if the line being typed in is a complete command, or the user just entered a newline to format the command better.

`info complete string \rm \\`**If `string` has no unmatched brackets, braces or parentheses, then a value of 1 is returned, else 0 is returned.**

Example

Example, part 1 (will produce an error):

```
set cmd "OK"
eval puts $cmd

set cmd "puts" ; lappend cmd {Also OK}; eval $cmd

set cmd "NOT OK"
eval puts $cmd
```

Result:

```
OK
Also OK
can not find channel named "NOT"
```

```
    while executing
"puts NOT OK"
    ("eval" body line 1)
    invoked from within
"eval puts $cmd"
    (file "xx.tcl" line 7)
```

Example, part 2:

```
eval [format {%s "%s"} puts "Even This Works"]

set cmd "And even this can be made to work"

eval [format {%s "%s"} puts $cmd ]

set tmpFileNum 0;

set cmd {proc tempFileName }
lappend cmd ""
lappend cmd "global num; incr num; return \"/tmp/TMP.[pid].\$num\""
eval $cmd

puts "\nThis is the body of the proc definition:"
puts "[info body tempFileName]\n"

set cmd {puts "This is Cool!}

if {[info complete $cmd]} {
    eval $cmd
} else {
    puts "INCOMPLETE COMMAND: $cmd"
}
```

Result:

```
Even This Works
And even this can be made to work

This is the body of the proc definition:
global num; incr num; return "/tmp/TMP.17520.$num"

INCOMPLETE COMMAND: puts "This is Cool!
```

7.3 Substitution without evaluation - format, subst

The Tcl interpreter does only one substitution pass during command evaluation. Some situations, such as placing the name of a variable in a variable, require two passes through the substitution phase. In this case, the `subst` command is useful.

`Subst` performs a substitution pass without performing any execution of commands except those required for the substitution to occur, ie: commands within `[]` will be executed, and the results placed in the return string.

The example code:

```
 puts "[subst $$c]\n"
```

shows an example of placing a variable name in a variable, and evaluating through the indirection.

The `format` command can also be used to force some levels of substitution to occur.

`subst ?-nobackslashes? ?-nocommands? ?-novariables? string`
Passes `string` through the Tcl substitution phase, and returns the original string with the backslash sequences, commands and variables replaced by their equivalents. If any of the `-no...` arguments are present, then that set of substitutions will not be done. **Note:** `subst` does not honor braces or quotes.

Example

```
set a "alpha"
set b a

puts {a and b with no substitution: $a $$b}
puts "a and b with one pass of substitution: $a $$b"
puts "a and b with subst in braces: [subst {$a $$b}]"
puts "a and b with subst in quotes: [subst "$a $$b"]\n"

puts "format with no subst [format {$%s} $b]"
puts "format with subst: [subst [format {$%s} $b]]"
eval "puts \"eval after format: [format {$%s} $b]\""

set num 0;
set cmd "proc tempFileName {} "
set cmd [format "%s {global num; incr num;" $cmd]
set cmd [format {%s return "/tmp/TMP.%s.$num"} $cmd [pid] ]
set cmd [format "%s }" $cmd ]
eval $cmd

puts "[info body tempFileName]"

set a arrayname
set b index
set c newvalue
eval [format "set %s(%s) %s" $a $b $c]

puts "Index: $b of $a was set to: $arrayname(index)"
```

Result:

```
a and b with no substitution: $a $$b
a and b with one pass of substitution: alpha $a
a and b with subst in braces: alpha $a
a and b with subst in quotes: alpha alpha

format with no subst $a
format with subst: alpha
eval after format: alpha
global num; incr num; return "/tmp/TMP.11168.$num"
Index: index of arrayname was set to: newvalue
```

7.4 Changing Working Directory - cd, pwd

Tcl also supports commands to change and display the current working directory.

These are:

`cd ?dirName?`
Changes the current directory to `dirName` (if `dirName` is given, or to the `\$HOME` directory if `dirName` is not given. If `dirName` is a tilde (~), `cd` changes the working directory to the user's home directory. If `dirName` starts with a tilde, then the rest of the characters are treated as a login ID, and `cd` changes the working directory to that user's $HOME.

`pwd`
Returns the current directory.

Example

```
set dirs [list TEMPDIR]

puts "[format "%-15s %-20s " "FILE" "DIRECTORY"]"

foreach dir $dirs {
    catch {cd $dir}
    set c_files [glob -nocomplain c*]

    foreach name $c_files {
        puts "[format "%-15s %-20s " $name [pwd]]"
    }
}
```

7.5 Debugging and Errors - errorInfo errorCode catch error return

In previous lessons we discussed how the `return` command could be used to return a value from a proc. In Tcl, a proc may return a value, but it always returns a status.

When a Tcl command or procedure encounters an error during its execution, the global variable `errorInfo` is set, and an error condition is generated. If you have proc `a` that called proc `b` that called `c` that called `d` , if `d` generates an error, the "call stack" will unwind. Since `d` generates an error, `c` will not complete execution cleanly, and will have to pass the error up to `b` , and in turn on to `a`. Each procedure adds some information about the problem to the report. For instance:

```
proc a {} {
    b
}
proc b {} {
    c
}
proc c {} {
    d
}
proc d {} {
    some_command
}

# Run the top-level proc
a
```

This produces the following output:

```
invalid command name "some_command"
    while executing
"some_command"
    (procedure "d" line 2)
    invoked from within
"d"
    (procedure "c" line 2)
    invoked from within
"c"
    (procedure "b" line 2)
    invoked from within
"b"
    (procedure "a" line 2)
    invoked from within
"a"
    (file "errors.tcl" line 16)
```

This actually occurs when any exception condition occurs, including `break` and `continue`. The `break` and `continue` commands normally occur within a loop of some sort, and the loop command catches the exception and processes it properly, meaning that it either stops executing the loop, or continues on to the next instance of the loop without executing the rest of the loop body.

It is possible to "catch" errors and exceptions with the `catch` command, which runs some code, and catches any errors that code happens to generate. The programmer can then decide what to do about those errors and act accordingly, instead of having the whole application come to a halt. A more flexible facility is the `try` command - see below.

For example, if an `open` call returns an error, the user could be prompted to provide another file name.

A Tcl proc can also generate an error status condition. This can be done by specifying an error return with an option to the `return` command, or by using the `error` command. In either case, a message will be placed in `errorInfo`, and the proc will generate an error.

`error message ?info? ?code?`
Generates an error condition and forces the Tcl call stack to unwind, with error information being added at each step. If `info` or `code` are provided, the errorInfo and errorCode variables are initialized with these values.

`catch script ?varName?`
Evaluates and executes `script`. The return value of `catch` is the status return of the Tcl interpreter after it executes `script` If there are no errors in `script`, this value is 0. Otherwise it is 1. If `varName` is supplied, the value returned by `script` is placed in `varName` if the script successfully executes. If not, the error is placed in `varName`.

`return ?-code code? ?-errorinfo info? ?-errorcode errorcode? ?value?`
Generates a return exception condition. The possible arguments are: `errorInfo`

Code	Description
`-code code`	The next value specifies the return status. ‘code‘ must be one of the following words: ‘ok‘ - Normal status return ‘error‘ - Proc returns error status ‘return‘ - Normal return ‘break‘ - Proc returns break status ‘continue‘ - Proc returns continue status These allow you to write procedures that behave like the built in commands `break`, `error`, and `continue`.
`-errorinfo info`	‘info‘ will be the first string in the `errorInfo` variable.
`-errorcode errorcode`	The proc will set `errorCode` to `errorcode`.
`value`	The string `value` will be the value returned by this proc.

`errorInfo` is a global variable that contains the error information from commands that have failed.

`errorCode`
`errorCode` is a global variable that contains the error code from command that

failed. This is meant to be in a format that is easy to parse with a script, so that Tcl scripts can examine the contents of this variable, and decide what to do accordingly.

Example

```
proc errorproc {x} {
    if {$x > 0} {
        error "Error generated by error" "Info String for error" $x
    }
}

catch errorproc
puts "after bad proc call: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

set errorInfo "";
catch {errorproc 0}
puts "after proc call with no error: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

catch {errorproc 2}
puts "after error generated in proc: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"


proc returnErr { x } {
    return -code error -errorinfo "Return Generates This" -errorcode "-999"
}

catch {returnErr 2}
puts "after proc that uses return to generate an error: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

proc withError {x} {
    set x $a
}

catch {withError 2}
puts "after proc with an error: ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"

catch {open [file join no_such_directory no_such_file] r}
puts "after an error call to a nonexistent file:"
puts "ErrorCode: $errorCode"
puts "ERRORINFO:\n$errorInfo\n"
```

Result:

```
after bad proc call: ErrorCode: TCL WRONGARGS
```

```
ERRORINFO:
wrong # args: should be "errorproc x"
    while executing
"errorproc"

after proc call with no error: ErrorCode: TCL WRONGARGS
ERRORINFO:


after error generated in proc: ErrorCode: 2
ERRORINFO:
Info String for error
    (procedure "errorproc" line 1)
    invoked from within
"errorproc 2"

after proc that uses return to generate an error: ErrorCode: -999
ERRORINFO:
Return Generates This
    invoked from within
"returnErr 2"

after proc with an error: ErrorCode: TCL READ VARNAME
ERRORINFO:
can’t read "a": no such variable
    while executing
"set x $a"
    (procedure "withError" line 2)
    invoked from within
"withError 2"

after an error call to a nonexistent file:
ErrorCode: POSIX ENOENT {no such file or directory}
ERRORINFO:
couldn’t open "no_such_directory/no_such_file": no such file or directory
    while executing
"open [file join no_such_directory no_such_file] r"
```

7.6 More Debugging - trace

When you are debugging Tcl code, sometimes it’s useful to be able to trace either the execution of the code, or simply inspect the state of a variable when various things happen to it. The `trace` command provides these facilities. It is a very powerful command that can be used in many interesting ways. It also risks being abused, and can lead to very difficult to understand code if it is used improperly (for instance, variables seemingly changing magically), so use it with care.

There are three principle operations that may be performed with the trace command:

- `add`, which has the general form: `trace add type ops ?args?`
- `info`, which has the general form: `trace info type name`
- `remove`, which has the general form:'trace remove type name opList command'

Which are for adding traces, retrieving information about traces, and removing traces, respectively. Traces can be added to three kinds of "things":

- `variable` - Traces added to variables are called when some event occurs to the variable, such as being written to or read.
- `command` - Traces added to commands are executed whenever the named command is renamed or deleted.
- `execution` - Traces on "execution" are called whenever the named command is run.

Traces on variables are invoked on four separate conditions - when a variable is accessed or modified via the `array` command, when the variable is read or written, or when it's unset. For instance, to set a trace on a variable so that when it's written to, the value doesn't change, you could do this:

```
proc vartrace {oldval varname element op} {
    upvar $varname localvar
    set localvar $oldval
}

set tracedvar 1
trace add variable tracedvar write [list vartrace $tracedvar]

set tracedvar 2
puts "tracedvar is $tracedvar"
```

In the above example, we create a proc that takes four arguments. We supply the first, the old value of the variable, because write traces are triggered **after** the variable's value has already been changed, so we need to preserve the original value ourselves. The other three arguments are the variable's name, the element name if the variable is an array (which it isn't in our example), and the operation to trace - in this case, `write`. When the trace is called, we simply set the variable's value back to its old value. We could also do something like generate an error, thus warning people that this variable shouldn't be written to. In fact, this would probably be better. If someone else is attempting to understand your program, they could become quite confused when they find that a simple `set` command no longer functions!

The command and execution traces are intended for expert users - perhaps those writing debuggers for Tcl in Tcl itself - and are therefore not covered in this tutorial, see the `trace` manual page for further information.

Example

```
proc traceproc {variableName arrayElement operation} {
    set op(write) Write
    set op(unset) Unset
    set op(read) Read

    set level [info level]
    incr level -1
    if {$level > 0} {
        set procid [info level $level]
    } else {
        set procid "main"
    }

    if {![string match $arrayElement ""]} {
        puts "TRACE: $op($operation) $variableName($arrayElement) in $procid"
    } else {
        puts "TRACE: $op($operation) $variableName in $procid"
    }
}

proc testProc {input1 input2} {
    upvar $input1 i
    upvar $input2 j

    set i 2
    set k $j
}

trace add variable i1 write traceproc
trace add variable i2 read traceproc
trace add variable i2 write traceproc

set i2 "testvalue"

puts "\ncall testProc"
testProc i1 i2

puts "\nTraces on i1: [trace info variable i1]"
puts "Traces on i2: [trace info variable i2]\n"

trace remove variable i2 read traceproc
puts "Traces on i2 after vdelete: [trace info variable i2]"

puts "\ncall testProc again"
testProc i1 i2
```

Result:

```
TRACE: Write i2 in main

call testProc
TRACE: Write i in testProc i1 i2
TRACE: Read j in testProc i1 i2

Traces on i1: {write traceproc}
Traces on i2: {write traceproc} {read traceproc}

Traces on i2 after vdelete: {write traceproc}

call testProc again
TRACE: Write i in testProc i1 i2
```

7.7 Command line arguments and environment strings

Scripts are much more useful if they can be called with different values in the command line.

For instance, a script may extract a particular value from a file by first asking for the name of the file, then reading the file name, and then extracting the data from that file.

An alternative is to use the arguments on the command-line: the script can then loop through all the files given there, and extract the data from each file.

The second method of writing the program can easily be used from other scripts. This is actually a very powerful method.

The number of command line arguments to a Tcl script is passed as the global variable `argc` . The name of a Tcl script is passed to the script as the global variable `argv0` , and the rest of the command line arguments are passed as a list in `argv`. The name of the executable that runs the script, such as `tclsh` is given by the command `info nameofexecutable`

Another method of passing information to a script is with **environment variables**. For instance, suppose you are writing a program in which a user provides some sort of comment to go into a record. It would be friendly to allow the user to edit their comments in their favorite editor. If the user has defined an `EDITOR` environment variable, then you can invoke that editor for them to use.

Environment variables are available to Tcl scripts in a global associative array `env` . The index into `env` is the name of the environment variable. The command `puts "\$env(PATH)"` would print the contents of the `PATH` environment variable.

Example

```
puts "There are $argc arguments to this script"
puts "The name of this script is $argv0"
if {$argc > 0} {puts "The other arguments are: $argv" }
```

```
puts "You have these environment variables set:"
foreach index [array names env] {
    puts "$index: $env($index)"
}
```

7.8 Timing scripts

The simplest method of making a script run faster is to buy a faster processor. Unfortunately, this isn't always an option. You may need to optimize your script to run faster. This is difficult if you can't measure the time it takes to run the portion of the script that you are trying to optimize.

The `time` command is the solution to this problem. `time` will measure the length of time that it takes to execute a script. You can then modify the script, rerun `time` and see how much you improved it.

`time script ?count?`
Returns the number of milliseconds it took to execute `script`. If `count` is specified, it will run the script `count` times, and average the result. The time is elapsed time, not CPU time.

Note: accurate timings of scripts require that you run them often enough for random variations in the computer's performance to average out. Therefore you should chose the repeat count with some care.

Example

```
proc timetst1 {lst} {
    set x [lsearch $lst "5000"]
    return $x
}

proc timetst2 {array} {
    upvar $array a
    return $a(5000);
}

# Make a long list and a large array.
for {set i 0} {$i < 5001} {incr i} {
    set array($i) $i
    lappend list $i
}

puts "Time for list search: [ time {timetst1 $list} 100]"
puts "Time for array index: [ time {timetst2 array} 100]"
```

Result:

```
Time for list search: 93.83 microseconds per iteration
Time for array index: 1.52 microseconds per iteration
```

7.9 Channel I/O: socket, fileevent, vwait

Tcl I/O is based on a the concept of channels. A channel is conceptually similar to a `FILE *` in C, or a stream in shell programming. The difference is that a channel may be a either a stream device like a file, or a connection oriented construct like a socket.

A stream based channel is created with the `open` command, as discussed in *lesson 26* (Section **??**). A socket based channel is created with a `socket` command. A socket can be opened as either as a client, or as a server.

If a socket channel is opened as a server, then the tcl program will 'listen' on that channel for another task to attempt to connect with it. When this happens, a new channel is created for that link (server-¿ new client), and the tcl program continues to listen for connections on the original port number. In this way, a single Tcl server could be talking to several clients simultaneously.

When a channel exists, a handler can be defined that will be invoked when the channel is available for reading or writing. This handler is defined with the `fileevent` command. When a tcl procedure does a `gets` or `puts` to a blocking device, and the device isn't ready for I/O, the program will block until the device is ready. This may be a long while if the other end of the I/O channel has gone off line. Using the `fileevent` command, the program only accesses an I/O channel when it is ready to move data.

Finally, there is a command to wait until an event happens. The `vwait` command will wait until a variable is set. This can be used to create a semaphore style functionality for the interaction between client and server, and let a controlling procedure know that an event has occurred.

Look at the example, and you'll see the `socket` command being used as both client and server, and the `fileevent` and `vwait` commands being used to control the I/O between the client and server.

`socket -server command ?options? port`
The `socket` command with the `-server` flag starts a server socket listing on port `port`. When a connection occurs on `port`, the proc `command` is called with the arguments:

- `channel` - The channel for the new client
- `address` - The IP Address of this client
- `port` The port that is assigned to this client

`socket ?options? host port`
The `socket` command without the `-server` option opens a client connection to the system with IP Address `host` and port address `port`. The IP Address may be given as a numeric string, or as a fully qualified domain address. To connect to the local host, use the address 127.0.0.1 (the loopback address).

`fileevent channelID writeable ?script?`
The `fileevent` command defines a handler to be invoked when a condition occurs. The conditions are `readable`, which invokes `script` when data is ready

to be read on `channelID`, and `writeable`, when `channelID` is ready to receive data. Note that end-of-file must be checked for by the `script`.

`vwait varName`
The `vwait` command pauses the execution of a script until some background action sets the value of `varName`. A background action can be a proc invoked by a fileevent, or a socket connection, or an event from a Tk widget.

Example

```
#
# Define two auxiliary procs
#
proc serverOpen {channel addr port} {
    global connected
    set connected 1
    fileevent $channel readable [list readLine Server $channel]
    puts "OPENED"
}

proc readLine {who channel} {
    global didRead
    if { [gets $channel line] < 0} {
        fileevent $channel readable {}
        after idle "close $channel;set out 1"
    } else {
        puts "READ LINE: $line"
        puts $channel "This is a return"
        flush $channel;
        set didRead 1
    }
}
```

The code to start the *server* and connect to it from a *client*:

```
set connected 0
# catch {socket -server serverOpen 33000} server
set server [socket -server serverOpen 33000]

after 100 update

set sock [socket -async 127.0.0.1 33000]
vwait connected

puts $sock "A Test Line"
flush $sock
vwait didRead
set len [gets $sock line]
puts "Return line: $len -- $line"
```

```
catch {close $sock}
vwait out
close $server
```

Result:

```
OPENED
READ LINE: A Test Line
Return line: 16 -- This is a return
```

7.10 Time and Date - clock

The `clock` command provides access to the time and date functions in Tcl. Depending on the subcommands invoked, it can acquire the current time, or convert between different representations of time and date.

The `clock` command is a platform independent method of getting the display functionality of the unix `date` command, and provides access to the values returned by a call to the UNIX system function `gettime()`.

`clock seconds`
The `clock seconds` command returns the time in seconds since the *epoch*. The date of the epoch varies for different operating systems, thus this value is useful for comparison purposes, or as an input to the `clock format` command.

`clock format clockValue ?-gmt boolean? ?-format string?`
The `format` subcommand formats a `clockValue` (as returned by `clock seconds`) into a human readable string. The `-gmt` switch takes a boolean as the second argument. If the boolean is `1` or `true`, then the time will be formatted as Greenwich Mean Time, otherwise, it will be formatted as local time, taking care of daylight saving time and timezones.

The `-format` option controls what format the return will be in. The contents of the `string` argument to format has similar contents as the format statement. However, the `\%*` descriptors are dedicated to dates and times:

`clock scan dateString ?-format format?`
The `scan` subcommand converts a human readable string to a system clock value, as would be returned by `clock seconds`.

The `dateString` argument for the `clock scan` command, without the `-format format` option, may be a string in any of these forms:

- time: A time of day in one of the formats shown below. Meridian may be AM, or PM, or a capitalization variant. If it is not specified, then the hour (hh) is interpreted as a 24 hour clock. Zone may be a three letter description of a time zone, EST, PDT, etc.

  * hh:mm:ss ?meridian? ?zone?

** hhmm ?meridian? ?zone?

- date: A date in one of the formats shown below.

Code	Meaning
`\%a`	Abbreviated weekday name (Mon, Tue, etc.)
`\%A`	Full weekday name (Monday, Tuesday, etc.)
`\%b`	Abbreviated month name (Jan, Feb, etc.)
`\%B`	Full month name (January, February, etc.)
`\%d`	Day of month
`\%j`	Julian day of year
`\%m`	Month number (01-12)
`\%y`	Year in century
`\%Y`	Year with 4 digits
`\%H`	Hour (00-23)
`\%I`	Hour (00-12)
`\%M`	Minutes (00-59)
`\%S`	Seconds(00-59)
`\%p`	PM or AM
`\%D`	Date as %m/%d/%y
`\%r`	Time as %I:%M:%S %p
`\%R`	Time as %I:%M
`\%T`	Time as %I:%M:%S
`\%Z`	Time Zone Name

** mm/dd/yy ** mm/dd ** monthname dd, yy ** monthname dd ** dd monthname yy ** dd monthname ** day, dd monthname yy

This is known as "free form scanning". The `clock scan` command then uses a number of heuristic rules to determine the correct form. However, it is safer to explicitly specify a format to which the date/time string should conform. This way you can be sure that the date/time is interpreted in a controlled way.

A small example of the use of the `-format` option with `clock scan`: in German speaking countries, the date is often formatted as *2017.01.02*, meaning the second of January, 2017. This is not in a form accepted by the free form scanning rules:

```
% puts [clock scan "2017.01.02" -format "%Y.%m.%d"]
1483311600
% puts [clock format 1483311600]
Mon Jan 02 00:00:00 CET 2017
```

Arithmetic with the `clock` command is also possible:

```
# Calculate the date ten days and 2 hours from today
set today [clock seconds]

set 10daysFromNow [clock add $today 10 days 2 hours]
puts [clock format $today]
puts [clock format $10daysFromNow]
```

might print:

```
Thu Jun 29 19:48:27 CEST 2017
Sun Jul 09 21:48:27 CEST 2017
```

(Note the time zone: the dates fall within the daylight saving period of the year 2017)

Other subcommands of the `clock` command are useful to measure time in shorter units than a second: `clock milliseconds` and `clock microseconds`.

Example

```
set systemTime [clock seconds]

puts "The time is: [clock format $systemTime -format %H:%M:%S]"
puts "The date is: [clock format $systemTime -format %D]"
puts [clock format $systemTime -format {Today is: %A, the %d of %B, %Y}]
puts "\nThe default format for the time is: [clock format $systemTime]\n"

set halBirthBook "Jan 12, 1997"
set halBirthMovie "Jan 12, 1992"
set bookSeconds [clock scan $halBirthBook]
set movieSeconds [clock scan $halBirthMovie]

puts "The book and movie versions of ’2001, A Space Odyssey’ had a"
puts "discrepancy of [expr {$bookSeconds - $movieSeconds}] seconds in how"
puts "soon we would have sentient computers like the HAL 9000"
```

Result:

```
The time is: 19:51:19
The date is: 06/29/2017
Today is: Thursday, the 29 of June, 2017

The default format for the time is: Thu Jun 29 19:51:19 CEST 2017

The book and movie versions of ’2001, A Space Odyssey’ had a
discrepancy of 157852800 seconds in how
soon we would have sentient computers like the HAL 9000
```

¡¡enddiscussion¿

7.11 More channel I/O - fblocked and fconfigure

The previous lessons have shown how to use channels with files and blocking sockets. Tcl also supports non-blocking reads and writes, and allows you to configure the sizes of the I/O buffers, and how lines are terminated.

A non-blocking read or write means that instead of a `gets` call waiting until data is available, it will return immediately. If there was data available, it will be read, and if no data is available, the `gets` call will return a 0 length.

If you have several channels that must be checked for input, you can use the `fileevent` command to trigger reads on the channels, and then use the `fblocked` command to determine when all the data is read.

The `fblocked` and `fconfigure` commands provide more control over the behavior of a channel.

The `fblocked` command checks whether a channel has returned all available input. It is useful when you are working with a channel that has been set to non-blocking mode and you need to determine if there should be data available, or if the channel has been closed from the other end.

The `fconfigure` command has many options that allow you to query or fine tune the behavior of a channel including whether the channel is blocking or non-blocking, the buffer size, the end of line character, etc.

`fconfigure channel ?param1? ?value1? ?param2 value2 ...?`
Configures the behavior of a channel. If no `param` values are provided, a list of the valid configuration parameters and their values is returned.

If a single parameter is given on the command line, the value of that parameter is returned.

If one or more pairs of `param/value` pairs are provided, those parameters are set to the requested value.

Parameters that can be set include:

- `-blocking` - determines whether or not the task will block when data cannot be moved on a channel (i.e. if no data is available on a read, or the buffer is full on a write).

- `-buffersize` - The number of bytes that will be buffered before data is sent, or can be buffered before being read when data is received. The value must be an integer between 10 and 1000000.

- `-translation` - Sets how Tcl will terminate a line when it is output. By default, the lines are terminated with the newline, carriage return, or newline/carriage return that is appropriate to the system on which the interpreter is running. This can be configured to be:

** `auto` - Translates newline, carriage return, or newline/carriage return as an end of line marker. Outputs the correct line termination for the current platform. ** `binary` - Treats newlines as end of line markers. Does not add any line termination to lines being output. This option is also useful if the file or channel is to be treated as *binary.* ** `cr` - Treats carriage returns as the end of line marker (and translates them to newline internally). Output lines are terminated with a carriage return. This is the Apple standard. ** `crlf` - Treats cr/lf pairs as the end of line marker, and terminates output lines with a carriage return/linefeed combination. This is the Windows standard, and should also be used for all line-oriented network protocols. ** `lf` - Treats linefeeds as the end of line marker, and terminates output lines with a linefeed. This is the Unix standard.

The example is similar to the *example with a client and server socket* (Section 7.9) in the same script. It shows a server channel being configured to be non-blocking, and using the default buffering style - data is not made available to the script until a newline is present, or the buffer has filled. When the first write:

```
puts -nonewline $sock "A Test Line"`
```

is done, the `fileevent` triggers the read, but the `gets` can't read characters because there is no newline. The `gets`returns a -1, and `fblocked` returns a 1.

When a bare newline is sent, the data in the input buffer will become available, and the `gets` returns `18`, and `fblocked` returns 0.

Example

We expand the two auxiliary procs, so that we get information about the state of the channel:

```
proc serverOpen {channel addr port} {
    puts "channel: $channel - from Address: $addr Port: $port"
    puts "The default state for blocking is: [fconfigure $channel -blocking]"
    puts "The default buffer size is: [fconfigure $channel -buffersize ]"

    # Set this channel to be non-blocking.
    fconfigure $channel -blocking 0
    set bl [fconfigure $channel -blocking]
    puts "After fconfigure the state for blocking is: $bl"

    # Change the buffer size to be smaller
    fconfigure $channel -buffersize 12
    puts "After Fconfigure buffer size is: [fconfigure $channel -buffersize ]\n"

    # When input is available, read it.
    fileevent $channel readable "readLine Server $channel"
}

proc readLine {who channel} {
    global didRead
    global blocked

    puts "There is input for $who on $channel"

    set len [gets $channel line]
    set blocked [fblocked $channel]
    puts "Characters Read: $len Fblocked: $blocked"

    if {$len < 0} {
        if {$blocked} {
            puts "Input is blocked"
        } else {
            puts "The socket was closed - closing my end"
            close $channel;
        }
    } else {
        puts "Read $len characters: $line"
        puts $channel "This is a return"
        flush $channel;
    }
    incr didRead;
}
```

Now, start the server and send a few lines of text, while also manipulating the channel's state.

```
set server [socket -server serverOpen 33000]

after 120 update; # This kicks MS-Windows machines for this application

set sock [socket 127.0.0.1 33000]

set bl [fconfigure $sock -blocking]
set bu [fconfigure $sock -buffersize]
puts "Original setting for sock: Sock blocking: $bl buffersize: $bu"

fconfigure $sock -blocking No
fconfigure $sock -buffersize 8;

set bl [fconfigure $sock -blocking]
set bu [fconfigure $sock -buffersize]
puts "Modified setting for sock: Sock blocking: $bl buffersize: $bu\n"

# Send a line to the server -- NOTE flush

set didRead 0
puts -nonewline $sock "A Test Line"
flush $sock;

# Loop until two reads have been done.

while {$didRead < 2} {
    # Wait for didRead to be set
    vwait didRead
    if {$blocked} {
        puts $sock "Newline"
        flush $sock
        puts "SEND NEWLINE"
    }
}

set len [gets $sock line]
puts "Return line: $len -- $line"
close $sock
vwait didRead
catch {close $server}
```

Result:

```
Original setting for sock: Sock blocking: 1 buffersize: 4096
Modified setting for sock: Sock blocking: 0 buffersize: 8

channel: sock0000000001D55EE0 - from Address: 127.0.0.1 Port: 63464
The default state for blocking is: 1
The default buffer size is: 4096
After fconfigure the state for blocking is: 0
After Fconfigure buffer size is: 12

There is input for Server on sock0000000001D55EE0
Characters Read: -1 Fblocked: 1
Input is blocked
SEND NEWLINE
There is input for Server on sock0000000001D55EE0
Characters Read: 18 Fblocked: 0
Read 18 characters: A Test LineNewline
Return line: 16 -- This is a return
There is input for Server on sock0000000001D55EE0
Characters Read: -1 Fblocked: 0
The socket was closed - closing my end
```

7.12 Child interpreters

For most applications, a single interpreter and subroutines are quite sufficient. However, if you are building a client-server system (for example) you may need to have several interpreters talking to different clients, and maintaining their state. You can do this with state variables, naming conventions, or swapping state to and from disk, but that gets messy. The `interp` command creates new child interpreters within an existing interpreter. The child interpreters can have their own sets of variables, commands and open files, or they can be given access to items in the parent interpreter.

If the child is created with the `-safe` option, it will not be able to access the file system, or otherwise damage your system. This feature allows a script to evaluate code from an unknown (and untrusted) source.

The names of child interpreters are a hierarchical list. If interpreter `foo` is a child of interpreter `bar`, then it can be accessed from the toplevel interpreter as `{bar foo}`.

The primary interpreter (what you get when you type tclsh) is the empty list `{}`.

The `interp` command has several subcommands and options. A critical subset is:

`interp create -safe name`
Creates a new interpreter and returns the name. If the `-safe` option is used, the new interpreter will be unable to access certain dangerous system facilities.

`interp delete name`
Deletes the named child interpreter.

`interp eval name args`
This is similar to the regular `eval` command, except that it evaluates the script in the child interpreter `name` instead of the primary interpreter. The `interp eval` command concatenates the args into a string, and ships that line

to the child interpreter to evaluate.

`interp alias srcPath srcCmd targetPath targetCmd ?arg ...?`
The `interp alias` command allows a script to share procedures between child interpreters or between a child and the primary interpreter.

Note that slave interpreters have a separate state and namespace, but do *not* have separate event loops. These are not threads, and they will not execute independently. If one slave interpreter gets stopped by a blocking I/O request, for instance, no other interpreters will be processed until it has unblocked.

The example below shows two child interpreters being created under the primary interpreter `{}`. Each of these interpreters is given a variable `name` which contains the name of the interpreter.

Note that the alias command causes the procedure to be evaluated in *the interpreter in which the procedure was defined*, not the interpreter in which it is evaluated. If you need a procedure to exist within an interpreter, you must `interp eval` a `proc` command within that interpreter. If you want an interpreter to be able to call back to the primary interpreter (or other interpreter) you can use the `interp alias` command.

Example

```
set i1 [interp create firstChild]
set i2 [interp create secondChild]

puts "first child interp: $i1"
puts "second child interp: $i2\n"

# Set a variable "name" in each child interp, and
# create a procedure within each interp
# to return that value
foreach int [list $i1 $i2] {
    interp eval $int [list set name $int]
    interp eval $int {proc nameis {} {global name; return "nameis: $name";} }
}

foreach int [list $i1 $i2] {
    interp eval $int "puts \"EVAL IN $int: name is \$name\""
    puts "Return from 'nameis' is: [interp eval $int nameis]"
}

#
# A short program to return the value of "name"
#
proc rtnName {} {
    global name
    return "rtnName is: $name"
}

#
```

```
# Alias that procedure to a proc in $i1
interp alias $i1 rtnName {} rtnName

puts ""

# This is an error. The alias causes the evaluation
# to happen in the {} interpreter, where name is
# not defined.
puts "firstChild reports [interp eval $i1 rtnName]"
```

Result:

```
first child interp: firstChild
second child interp: secondChild

EVAL IN firstChild: name is firstChild
Return from 'nameis' is: nameis: firstChild
EVAL IN secondChild: name is secondChild
Return from 'nameis' is: nameis: secondChild

can't read "name": no such variable
    while executing
"return "rtnName is: $name""
    (procedure "rtnName" line 3)
    invoked from within
"rtnName"
    invoked from within
"interp eval $i1 rtnName"
    invoked from within
"puts "firstChild reports [interp eval $i1 rtnName]""
    (file "xx.tcl" line 37)
```