Preface / 前言

This tutorial is based on Tcl Tutorial. This tutorial was originally designed for the lab of the course ECE4810J. This tutorial only keeps the key notes and adds some additional instructions and explanations. The chapters are also re-organized. This tutorial assumes that readers have basic knowledge of C/C++ programming concepts and shell programing concepts, otherwise to read the original full tutorial is recommended. This tutorial is written by Yihua Liu.
@TOC

Running Tcl

Tcl/Tk Software

Simple Text Output

`puts` : output a string
`;` : end of the command
`""` : enclose a string
`{}` : enclose a string
`#` : comment at a beginning of a line
`;#` : comment after the command

```
% puts HelloWorld
HelloWorld
% puts "This is line 1"; puts "this is line 2"
This is line 1
this is line 2
% puts {Hello, World - In Braces}
Hello, World - In Braces
% # This is a comment at beginning of a line
% puts "Hello, World - In quotes"    ;# This is a comment after the command.
Hello, World - In quotes
% puts "Hello, World; - With  a semicolon inside the quotes"
Hello, World; - With  a semicolon inside the quotes
```

Assigning values to variables

`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.
`$`*varName*: use the value of the variable

```
% set Y 1.24
1.24
% puts $Y
1.24
% set label "The value in Y is:"
The value in Y is:
% puts "$label $Y"
The value in Y is: 1.24
```

Evaluation & Substitutions

Grouping arguments with ""

Grouping words with double quotes allows substitutions to occur within the double quotes.
`\` : escape characters; line feed of a single command
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.

```
% puts "This string comes out\
on a single line"
This string comes out on a single line
```

Grouping arguments with {}

Grouping words within double braces **disables** substitution within the braces.
`\` : line feed of a single command

```
% set Z Albany
Albany
% set Z_LABEL "The Capitol of New York is:"
The Capitol of New York is:
% puts "$Z_LABEL $Z"
The Capitol of New York is: Albany
% puts {$Z_LABEL $Z}
$Z_LABEL $Z
% puts {There are no substitutions done within braces \n \r \x0a \f \v}
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}
But, the escaped newline at the end of a string is still evaluated as a space
```

Nesting braces and double quotes:

```
% puts "$Z_LABEL {$Z}"
The Capitol of New York is:  {Albany}
```

Grouping arguments with []

`[]` : the results of a command

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

Remember that double quotes do substitution while curly braces do not:

```
% set z {[set x "This is a string within quotes within braces"]}
[set x "This is a string within quotes within braces"]
% puts "Note the curly braces: $z\n"
Note the curly braces: [set x "This is a string within quotes within braces"]
% set a "[set x {This is a string within braces within quotes}]"
This is a string within braces within quotes
% puts "See how the set is executed: $a"
See how the set is executed: This is a string within braces within quotes
% set b "\[set y {This is a string within braces within quotes}]"
[set y {This is a string within braces within quotes}]
% puts "Note the \\ escapes the bracket:\n\$b is: $b"  ;# variable y does not exist
Note the \ escapes the bracket:
$b is: [set y {This is a string within braces within quotes}]
```

Results of a command - Math 101

`expr` : doing math type operations
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`.
WARNING: You should **always** use braces when evaluating expressions that may contain user input, to avoid possible security breaches. The `expr` command performs its own round of substitutions on variables and commands, so you should use braces to prevent the Tcl interpreter doing this as well (leading to double substitution).

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

Operands

Octal numbers: 0700
Hexadecimal numbers: 0x32
Floating-point numbers: 2.1
Omit trailing zero(s): 3.
Big E: 6E4
Small e: 7.91e+16
Omit 0 on ones unit: .000001

Operators

Sign	Operator

Sign	Operator
-	subtract/unary minus
+	add/unary plus
~	bit-wise NOT (integers only)
!	logical NOT
**	exponentiation
*	multiply
/	divide
%	remainder (integers only)
<<	left (bit) shift (integers only)
>>	right (bit) shift (integers only)
<	less (applicable to non-numeric strings)
>	greater (applicable to non-numeric strings)
<=	less than or equal (applicable to non-numeric strings)
>=	greater than or equal (applicable to non-numeric strings)
&	bit-wise AND (integers only)
^	bit-wise exclusive OR (integers only)
|	bit-wise OR (integers only)
&&	logical AND
||	logical OR
x?y:z	if-then-else

All the operators above can evaluate numeric strings as well as numbers:

```
% expr {1+2}
3
% expr {"1"+"2"}
3
% expr {"1"<<"2"}
4
```

<, >, <=, and >= are relational operators that can do string comparisons as well, and produce 1 if the condition is true, 0 otherwise.

```
% expr {"a"<"b"}
1
% expr {"ba"<"b"}
0
```

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

Math functions

`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`

Type conversions

- `double()` converts a number to a 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 same as `int()` or `wide()` or an integer of arbitrary size (in Tcl 8.5 and above).

Working with arrays:

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

Example: the trigonometric functions

```
% set pi6 [expr {3.1415926/6.0}]
0.5235987666666667
% puts "The sine and cosine of pi/6: [expr {sin($pi6)}] [expr {cos($pi6)}]"
The sine and cosine of pi/6: 0.49999999226497965 0.8660254082502546
```

`string length`

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

`string length` returns the number of characters in a value.
Synopsis:
`string length` *string*

Computers and numbers

This chapter behaves very differently from Tcl 8.4 or before.

```
% puts [expr {1000000*1000000}]
1000000000000
% puts [expr {1000000*1000000.}]
1000000000000.0
% puts [expr {1.0e300/1.0e-300}]
Inf
% puts "1/2 is [expr {1/2}]"
1/2 is 0
% puts "-1/2 is [expr {-1/2}]"
-1/2 is -1
% puts "1/2 is [expr {1./2}]"
1/2 is 0.5
% puts "1/3 is [expr {1./3}]"
1/3 is 0.3333333333333333
% puts "1/3 is [expr {double(1)/3}]"
1/3 is 0.3333333333333333
```

a = q * b + r
0 <= |r| < |b|
r has the same sign as q
Show all decimals needed to exactly reproduce a particular number:

```
% set tcl_precision 17
17
% puts "1/2 is [expr {1./2}]"
1/2 is 0.5
% puts "1/3 is [expr {1./3}]"
1/3 is 0.33333333333333331
% set a [expr {1.0/3.0}]
0.33333333333333331
% puts "3*(1/3) is [expr {3.0*$a}]"
3*(1/3) is 1.0
% set b [expr {10.0/3.0}]
3.3333333333333335
% puts "3*(10/3) is [expr {3.0*$b}]"
3*(10/3) is 10.0
% set c [expr {10.0/3.0}]
3.3333333333333335
% set d [expr {2.0/3.0}]
0.66666666666666663
% puts "(10.0/3.0) / (2.0/3.0) is [expr {$c/$d}]"
(10.0/3.0) / (2.0/3.0) is 5.0000000000000009
% set e [expr {1.0/10.0}]
```

```
0.10000000000000001
% puts "1.2 / 0.1 is [expr {1.2/$e}]"
1.2 / 0.1 is 11.999999999999998
% set number [expr {int(1.2/0.1)}]
11
```

Transcendental functions are not standardised:

```
% set pi1 [expr {4.0*atan(1.0)}]
3.1415926535897931
% set pi2 [expr {6.0*asin(0.5)}]
3.1415926535897936
% puts [expr {$pi1-$pi2}]
-4.4408920985006262e-16
```

Do not rely on formulae to obtain the π value.

Numeric Comparisons 101 - if

`if`

Synopsis:
`if` *expr1* ?then? *body1* elseif *expr2* ?then? *body2* elseif ... ?else? *?bodyN?*

```
% set x 1
1
% if {$x == 2} {puts "$x is 2"} else {puts "$x is not 2"}
1 is not 2
% if {$x != 1} {
    puts "$x is != 1"
} else {
    puts "$x is 1"
}
1 is 1
% if $x==1 {puts "GOT 1"}
GOT 1
```

A dangerous example: due to the extra round of substitution, the script stops:

```
% set y {[exit]}
[exit]
% if "$$y != 1" {
    puts "$$y is != 1"
} else {
    puts "$$y is 1"
}
invalid character "$"
in expression "$[exit] != 1"
```

`exit`

Synopsis:
`exit` *?returnCode?*
`exit` terminates the entire process, not just the interpreter it is is executed from. *returnCode*, which defaults to 0, is the exit status to report. A non-zero exit status is usually interpreted as an error case by the calling process, the exit command useful for signaling that the process did not complete normally.

Textual Comparison - switch

Synopsis:
`switch` *string pattern1 body1 ?pattern2 body2? ... ?patternN bodyN?*
`switch` *string* { *pattern1 body1 ?pattern2 body2? ... ?patternN bodyN?* }
If the last *pattern* argument is the string `default`, that pattern will match any string.
If there is no `default` argument, and none of the *patterns* match *string*, then the `switch` command will return an empty string.

```
% set x "ONE"
ONE
% set y 1
1
% set z ONE
ONE
% 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"
    }
}
MATCH ONE. 1 + one is 2
% switch $x "ONE" "puts ONE=1" "TWO" "puts TWO=2" "default" "puts NO_MATCH"
ONE=1
% switch $x \
"ONE"     "puts ONE=1"  \
"TWO"     "puts TWO=2" \
"default"     "puts NO_MATCH";
ONE=1
```

If you use the brace version of this command, there will be no substitutions done on the patterns. The body of the command, however, will be parsed and evaluated just like any other command.

Looping 101 - While loop

`while`

Synopsis:
`while` *test body*
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!

```
% set x 1
1
% while {$x < 5} {
    puts "x is $x"
    set x [expr {$x + 1}]
}
x is 1
x is 2
x is 3
x is 4
% puts "exited first loop with X equal to $x"
exited first loop with X equal to 5
% set x 0
0
% while "$x < 5" {
    set x [expr {$x + 1}]
    if {$x > 7} break
    if "$x > 3" continue
    puts "x is $x"
}
x is 1
x is 2
x is 3
% puts "exited second loop with X equal to $x"
exited second loop with X equal to 8
```

Why? Because for the double quoted *test*, the substitution of `x` in the *test* is determined in the beginning and not to be updated during the loop, while the substitution of `x` in the *body* is updated during the loop. The value of `x` in the *test* is always 0, so the evaluation of `"$x"<5` will always be true.

`break`

When a `break` is encountered, the loop exits immediately.

`continue`

When a `continue` is encountered, evaluation of the body ceases, and the test is re-evaluated.

Looping 102 - For and incr

`for`

Synopsis:
`for` *start test next body*
During evaluation of the for command, *start* is evaluated once and first, and then *test*. If true, then the *body* is evaluated, and finally the *next*. After that, the interpreter loops back to the *test*, and repeats the process. If the *test* evaluates as false, then the loop will exit immediately.
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

`incr`

`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.

```
% set i 0
0
% incr i
1
% set i [expr {$i + 1}]
2
% for {set i 0} {$i < 3} {incr i} {
    puts "I inside first loop: $i"
}
I inside first loop: 0
I inside first loop: 1
I inside first loop: 2
```

Adding new commands to Tcl - proc

Synopsis:
`proc` *name args body*
`return`: return the value of the body of a proc
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.

```
% proc sum {arg1 arg2} {
    set x [expr {$arg1 + $arg2}];
    return $x
}
% puts " The sum of 2 + 3 is: [sum 2 3]"
 The sum of 2 + 3 is: 5
% proc for {a b c} {
    puts "The for command has been replaced by a puts";
    puts "The arguments were: $a\n$b\n$c"
}
% for {set i 1} {$i < 10} {incr i}
The for command has been replaced by a puts
The arguments were: set i 1
$i < 10
incr i
```

Variations in proc arguments and return values

Default values:

```
% proc justdoit {a {b 1} {c -1}} {}
% justdoit 10  ;# a = 10, b = 1, c = -1
% justdoit 10 20  ;# a = 10, b = 20, c = -1
```

Variable arguments:
A proc will accept a variable number of arguments if the last declared argument is the word `args`.
Note that if there is a variable other than `args` after a variable with a default, then the default will never be used. For `proc function { a {b 1} c} {...}`, you will always have to call it with 3 arguments. Yet, the command below can accept a variable number of arguments:

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

```
% 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]
There is only one argument and it is: ONE
1
% set count2 [example ONE TWO]
There are two arguments - ONE and TWO
2
% set count3 [example ONE TWO THREE ]
There are many arguments - ONE and TWO and THREE
many
% set count4 [example ONE TWO THREE FOUR]
There are many arguments - ONE and TWO and THREE FOUR
many
% puts "The example was called with $count1, $count2, $count3, and $count4 Arguments"
The example was called with 1, 2, many, and many Arguments
```

Variable scope - global and upvar

`global`

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

`upvar`

The `upvar` command "ties" the name of a variable in the current scope to a variable in a different scope.
Synopsis:
`upvar` *?level? otherVar1 myVar1 ?otherVar2 myVar2? ... ?otherVarN myVarN?*
By default `level` is 1, the next level up.
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.

```
% proc SetPositive { variable value } {
    upvar $variable myvar
    if {$value < 0} {
        set myvar [expr {-$value}]
    } else {
        set myvar $value
    }
    return $myvar
}
% SetPositive x 5
5
% SetPositive y -5
5
% proc two {y} {
    upvar 1 $y z                    ;# Tie the calling value to variable z
    upvar 2 x a                     ;# Tie variable x two levels up to a
    puts "two: Z: $z A: $a"         ;# Output the values, just to confirm
    set z 1                         ;# Set z, the passed variable to 1;
    set a 2                         ;# Set x, two layers up to 2;
}
% proc one {y} {
    upvar $y z                      ;# This ties the calling value to variable z
    puts "one: Z: $z"               ;# Output that value, to check it is 5
    two z                           ;# Call proc two, which will change the value
}
% one y                             ;# Call one, and output X and Y after the call.
one: Z: 5
two: Z: 5 A: 5
2
% puts "X: $x  Y: $y"
X: 2  Y: 1
% proc existence {variable} {
    upvar $variable testVar
    if { [info exists testVar] } {
        puts "$variable Exists"
```

```
    } else {
        puts "$variable Does Not Exist"
    }
}
% set x 1
1
% set y 2
2
% for {set i 0} {$i < 5} {incr i} {
    set a($i) $i;
}
% existence x
x Exists
% unset x
% existence x
x Does Not Exist
```

`info exists`

Synopsis:
`info exists` *varName*
`info exists` tests whether a variable exists and is defined.
Returns 1 if the variable named *varName* exists in the current context according the the rules of name resolution, and has been defined by being given a value, returns 0 otherwise.

`unset`

Synopsis:
`unset` ?-nocomplain? *varName ?varName ...?*
Unsetting an upvar-bound variable will also unset all its other bindings:

```
% set var1 one
one
% proc p1 {} {
    upvar 1 var1 var1
    unset var1
}
% puts [info exists var1] ;# -> 1
1
% p1
% puts [info exists var1] ;# -> 0
0
```

`array set`

`array set` sets values in an array.
Synopsis:
`array set` *arrayName dictionary*

```
% array set val [list a 1 c 6 d 3]
% puts $val(a)
1
% puts $val(c)
6
```

Note that `array set` does not remove variables which already exist in the array.

`array unset`

`array unset` unsets values in an array.
Synopsis:
`array unset` *arrayName ?pattern?*
Unsets all of the variables in the array that match *pattern* (using the matching rules of string match).

Tcl Data Structures 101 - The list

The list

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"]`

Commands:

- `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.
- `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.

```
% set x "a b c"
a b c
% puts "Item at index 2 of the list {$x} is: [lindex $x 2]"
Item at index 2 of the list {a b c} is: c
% set y [split 7/4/1776 "/"]
7 4 1776
% puts "We celebrate on the [lindex $y 1]'th day of the [lindex $y 0]'th month"
We celebrate on the 4'th day of the 7'th month
% set z [list puts "arg 2 is $y" ]
puts {arg 2 is 7 4 1776}
% puts "A command resembles: $z"
A command resembles: puts {arg 2 is 7 4 1776}
% set i 0
0
% foreach j $x {
    puts "$j is item number $i in list x"
    incr i
}
a is item number 0 in list x
b is item number 1 in list x
c is item number 2 in list x
```

Adding & Deleting members of a list

- `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` *listName ?arg1 arg2 ... argn?*
  Appends the *args* to the list *listName* treating each *arg* as a list element.
- `linsert` *listName index arg1 ?arg2 ... argn?*
  Returns a new list with the new list elements inserted just before the *index* th element of *listName*. 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` *listName 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`.

```
% set b [list a b {c d e} {f {g h}}]
a b {c d e} {f {g h}}
% puts "Treated as a list: $b"
Treated as a list: a b {c d e} {f {g h}}
% set b [split "a b {c d e} {f {g h}}"]
a b \{c d e\} \{f \{g h\}\}
% puts "Transformed by split: $b"
Transformed by split: a b \{c d e\} \{f \{g h\}\}
% set a [concat a b {c d e} {f {g h}}]
a b c d e f {g h}
% puts "Concated: $a"
Concated: a b c d e f {g h}
% lappend a {ij K lm}                        ;# Note: {ij K lm} is a single element
```

```
a b c d e f {g h} {ij K lm}
% puts "After lappending: $a"
After lappending: a b c d e f {g h} {ij K lm}
% set b [linsert $a 3 "1 2 3"]               ;# "1 2 3" is a single element
a b c {1 2 3} d e f {g h} {ij K lm}
% puts "After linsert at position 3: $b"
After linsert at position 3: a b c {1 2 3} d e f {g h} {ij K lm}
% set b [lreplace $b 3 5 "AA" "BB"]
a b c AA BB f {g h} {ij K lm}
% puts "After lreplacing 3 positions with 2 values at pos 3: $b"
After lreplacing 3 positions with 2 values at pos 3: a b c AA BB f {g h} {ij K lm}
```

More list commands - lsearch, lsort, lrange

- `lsearch` *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.
- `lsort` *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.

```
% set list [list {Washington 1789} {Adams 1797} {Jefferson 1801} \
               {Madison 1809} {Monroe 1817} {Adams 1825} ]
{Washington 1789} {Adams 1797} {Jefferson 1801} {Madison 1809} {Monroe 1817} {Adams 1825}
% set x [lsearch $list Washington*]
0
% set y [lsearch $list Madison*]
3
% incr x
1
% incr y -1                        ;# Set range to be not-inclusive
2
% set subsetlist [lrange $list $x $y]
{Adams 1797} {Jefferson 1801}
% puts "The following presidents served between Washington and Madison"
The following presidents served between Washington and Madison
% foreach item $subsetlist {
    puts "Starting in [lindex $item 1]: President [lindex $item 0] "
}
Starting in 1797: President Adams
Starting in 1801: President Jefferson
% set x [lsearch $list Madison*]
3
% set srtlist [lsort $list]
{Adams 1797} {Adams 1825} {Jefferson 1801} {Madison 1809} {Monroe 1817} {Washington 1789}
% set y [lsearch $srtlist Madison*]
3
% puts "$x Presidents came before Madison chronologically"
3 Presidents came before Madison chronologically
% puts "$y Presidents came before Madison alphabetically"
3 Presidents came before Madison alphabetically
```

Simple pattern matching - "globbing"

Wildcards

- *****
  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.

```
% string match f* foo
1
% string match f?? foo
1
% string match f foo
0
% set bins [glob /usr/bin/*]
no files matched glob pattern "/usr/bin/*"
```

`string match`

Synopsis:
`string match` *?-nocase? pattern string*
Determine whether *pattern* matches *string*, returning return 1 if it does, 0 if it doesn't. If **-nocase** is specified, then the pattern attempts to match against the string in a case insensitive manner.

```
% string match *\\* "thistest \\" ;# -> true
0
% string match {\[} {[}
1
% string match \\\[ \[
1
```

Something trickier:

```
% string match {[AZ-]X]} A
0
% string match {[AZ-]X]} AX]
1
% string match a\\ a\\
0
% string match {a[\]} a\\
1
% string match a\\\\ a\\
1
```

String

String Subcommands - length index range

- `string index` *string index*
  Returns the *index_th character from _string*.
- `string range` *string first last*
  Returns a string composed of the characters from *first* to *last*.

```
% set string "this is my test string"
this is my test string
% puts "There are [string length $string] characters in \"$string\""
There are 22 characters in "this is my test string"
% puts "[string index $string 1] is the second character in \"$string\""
h is the second character in "this is my test string"
% puts "\"[string range $string 5 10]\" are characters between the 5'th and 10'th"
"is my " are characters between the 5'th and 10'th
```

String comparisons - compare match first last wordend

- `string compare` *string1 string2*
  Compares *string1* to *string2* and returns:
- -1 ... If *string1* is less than *string2*
- 0 ... If *string1* is equal to *string2*
- 1 ... If *string1* is greater than *string2*
  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` *string index*
  Returns the index of the character just after the last one in the word which contains the *index*'th character of *string*. A word is any contiguous set of letters, numbers or underscore characters, or a single other character.
- `string wordstart` *string index*
  Returns the index of the first character in the word that contains the *index*'th character of *string*. A word is any contiguous set of letters, numbers or underscore characters, or a single other character.

```
```

```
% set fullpath "/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17"
/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17
% set relativepath "CVS/Entries"
CVS/Entries
% set directorypath "/usr/bin/"
/usr/bin/
% set paths [list $fullpath $relativepath $directorypath]
/usr/home/clif/TCL_STUFF/TclTutor/Lsn.17 CVS/Entries /usr/bin/
% foreach path $paths  {
    set first [string first "/" $path]
    set last [string last "/" $path]
    if {$first != 0} {
        puts "$path is a relative path"
    } else {
        puts "$path is an absolute path"
    }
    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"
    }
    if {[string match "*CVS*" $path]} {
        puts "$path is part of the source code control tree"
    }
    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"
    }
}
```

Modifying Strings - tolower, toupper, trim, format

- `string tolower` *string*
  Returns *string* with all the letters converted from upper to lower case.
- `string toupper` *string*
  Returns *string* with all the letters converted from lower to upper case.
- `string trim` *string ?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` *string ?trimChars?*
  Returns *string* with all occurrences of *trimChars* removed from the left. By default *trimChars* are whitespace (spaces, tabs, newlines)
- `string trimright` *string ?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. 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.

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

```
....This string has leading dots
% set both  "((this string is nested in parens )))"
((this string is nested in parens )))
% puts "tolower converts this: $upper"
tolower converts this: THIS IS A STRING IN UPPER CASE LETTERS
% puts "              to this: [string tolower $upper]"
              to this: this is a string in upper case letters
% puts "toupper converts this: $lower"
toupper converts this: this is a string in lower case letters
% puts "              to this: [string toupper $lower]"
              to this: THIS IS A STRING IN LOWER CASE LETTERS
% puts "trimright converts this: $trailer"
trimright converts this: This string has trailing dots ....
% puts "               to this: [string trimright $trailer .]"
               to this: This string has trailing dots
% puts "trimleft converts this: $leader"
trimleft converts this: ....This string has leading dots
% puts "               to this: [string trimleft $leader .]"
               to this: This string has leading dots
% puts "trim converts this: $both"
trim converts this: ((this string is nested in parens )))
% puts "           to this: [string trim $both "()"]"
           to this: this string is nested in parens
% set labels [format "%-20s %+10s " "Item" "Cost"]
Item                       Cost
% set price1 [format "%-20s %10d Cents Each" "Tomatoes" "30"]
Tomatoes                     30 Cents Each
% set price2 [format "%-20s %10d Cents Each" "Peppers" "20"]
Peppers                      20 Cents Each
% set price3 [format "%-20s %10d Cents Each" "Onions" "10"]
Onions                       10 Cents Each
% set price4 [format "%-20s %10.2f per Lb." "Steak" "3.59997"]
Steak                      3.60 per Lb.
% puts "Example of format:"
 Example of format:
% puts "$labels"
Item                       Cost
% puts "$price1"
Tomatoes                     30 Cents Each
% puts "$price2"
Peppers                      20 Cents Each
% puts "$price3"
Onions                       10 Cents Each
% puts "$price4"
Steak                      3.60 per Lb.
```

Regular Expressions

Regular Expressions 101

- `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*.
- **^**
  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
- **[\^...]**
  Matches any character *NOT* a member of the set of characters following the ^.
- **(...)**
  Groups a set of characters into a subSpec.

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

```
% set result [regexp {[a-z]+} $sample match]
1
% puts "Result: $result match: $match"
Result: 1 match: here
% set result [regexp {([A-Za-z]+) +([a-z]+)} $sample match sub1 sub2 ]
1
% puts "Result: $result Match: $match 1: $sub1 2: $sub2"
Result: 1 Match: Where there 1: Where 2: there
% regsub "way" $sample "lawsuit" sample2
1
% puts "New: $sample2"
New: Where there is a will, There is a lawsuit.
% puts "Number of words: [regexp -all {[^ ]+} $sample]"
Number of words: 9
```

More Examples Of Regular Expressions

Finding floating-point numbers in a line of text (no scientific notation):

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

```
% set pattern  {(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])}
(^|[ \t])([-+]?(\d+|\.\d+|\d+\.\d*))($|[^+-.])
% set examples {"1.0"    " .02"  "  +0."
               "1"      "+1"    " -0.0120"
               "+0000"  " - "   "+."
               "0001"   "0..2"  "++1"
               "A1.0B"  "A1"}
"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"
    }
}
>>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
```

- Text enclosed in a string:
  `regexp {(["'])[^"']*\1} $string enclosed_string`
- If a word occurs twice in the same line of text:

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

- Counting the open and close parentheses

```
% if { [regexp -all {\(} $string] != [regexp -all {\)} $string] } {
    puts "Parentheses unbalanced!"
}
```

- If at any point while scanning the string there are more close parentheses than open parentheses:

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

Arguments:

- `-all`
  Matches the expression multiple times and returns the total number of matches found. Any specific match variables contain only the last corresponding match.
- `inline`
  Instead of placing matches in variables, returns a list of matches. Together with `-all`, iteratively matches the expression, each time concatenating the match and any submatches individually with the list, and returns that list. With `-inline`, match variables may not be specified.

More Quoting Hell - Regular Expressions 102

- 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 ([) has no special meaning to either the Tcl substitution phase or the regular expression parser.

An escape can be either enclosing the phrase in braces, or placing a backslash before the escaped character. To pass a left bracket to the regular expression parser to evaluate as a range of characters takes 1 escape. To have the regular expression parser match a literal left bracket takes 2 escapes (one to escape the bracket in the Tcl substitution phase, and one to escape the bracket in the regular expression parsing). If you have the string placed within quotes, then a backslash that you wish passed to the regular expression parser must also be escaped with a backslash.
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}]
{/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"
}
/ is 17086 blocks
/news is 179824 blocks
/usr is 1249244 blocks
/var is 98190 blocks
```

Extracting a hexadecimal value:

```
% set line {Interrupt Vector?   [32(0x20)]}
Interrupt Vector?       [32(0x20)]
% regexp "\[^\t]+\t\\\[\[0-9]+\\(0x(\[0-9a-fA-F]+)\\)]" $line match hexval
1
% puts "Hex Default is: 0x$hexval"
Hex Default is: 0x20
```

Matching the special characters as if they were ordinary:

```
% set str2 "abc^def"
abc^def
% regexp "\[^a-f]*def" $str2 match
1
% puts "using \[^a-f] the match is: $match"
using [^a-f] the match is: ^def
% regexp "\[a-f^]*def" $str2 match
1
```

```
% puts "using \[a-f^] the match is: $match"
using [a-f^] the match is: abc^def
% regsub {\^} $str2 " is followed by: " str3
1
% puts "$str2 with the ^ substituted is: \"$str3\""
abc^def with the ^ substituted is: "abc is followed by: def"
% regsub "(\[a-f]+)\\^(\[a-f]+)" $str2 "\\2 follows \\1" str3
1
% puts "$str2 is converted to \"$str3\""
abc^def is converted to "def follows abc"
```

Debugging and Errors - errorInfo errorCode catch error return

`error`

- `error`
  Synopsis:
  `error` *message ?info? ?code?*
  Generates an error with the specified *message*. If supplied, info is used to seed the errorInfo value, and code becomes the errorCode, which otherwise is `NONE`.
  `error` is short for `return -level 0 -code error`, which is not the same as `return -code error`, the latter being the equivalent of `return -level 1 -code error`. With the `-level 0` variant, `-errorinfo` contains the line number that `return` was called at, whereas with the level 1 variant, errorinfo contains the line number in the caller that the current routine was called at. When in doubt, just use error.
- `errorInfo`
  `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.

`catch`

Synopsis:
`catch script` *?messageVarName? ?optionsVarName?*
`catch` is used to intercept the return code from the evaluation of *script*. The most common use case is probably just to ignore any error that occurred during the evaluation of `$script`.
`$messageVarName` contains the value that result from the evaluation of `$script`. When an exceptional return code is returned, `$messageVarName` contains the message corresponding to that exception.
The standard return codes are 0 to 4, as defined for `return`, and also in `tcl.h`.

`return`

Synopsis:
`return` *?-code code? ?-errorinfo info? ?-errorcode errorcode? ?value?*
Generates a return exception condition. The possible arguments are:

- `-code` *code*
  The next value specifies the return status. *code* must be one of:
  - `ok` - Normal status return
  - `error` - Proc returns error status
  - `return` - Normal return
  - `break` - Proc returns break status
  - `continue` - Proc returns continue status
- `-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.

```
% proc errorproc {x} {
    if {$x > 0} {
        error "Error generated by error" "Info String for error" $x
    }
}
% catch errorproc
1
% puts "after bad proc call: ErrorCode: $errorCode"
after bad proc call: ErrorCode: TCL WRONGARGS
% puts "ERRORINFO:\n$errorInfo"
ERRORINFO:
```

```
wrong # args: should be "errorproc x"
    while executing
"errorproc"
% set errorInfo "";
% catch {errorproc 0}
0
% puts "after proc call with no error: ErrorCode: $errorCode"
after proc call with no error: ErrorCode: TCL WRONGARGS
% puts "ERRORINFO:\n$errorInfo"
ERRORINFO:

% catch {errorproc 2}
1
% puts "after error generated in proc: ErrorCode: $errorCode"
after error generated in proc: ErrorCode: 2
% puts "ERRORINFO:\n$errorInfo"
ERRORINFO:
Info String for error
    (procedure "errorproc" line 1)
    invoked from within
"errorproc 2"
% proc returnErr { x } {
    return -code error -errorinfo "Return Generates This" -errorcode "-999"
}
% catch {returnErr 2}
1
% puts "after proc that uses return to generate an error: ErrorCode: $errorCode"
after proc that uses return to generate an error: ErrorCode: -999
% puts "ERRORINFO:\n$errorInfo"
ERRORINFO:
Return Generates This
    invoked from within
"returnErr 2"
% proc withError {x} {
    set x $a
}
% catch {withError 2}
1
% puts "after proc with an error: ErrorCode: $errorCode"
after proc with an error: ErrorCode: TCL READ VARNAME
% puts "ERRORINFO:\n$errorInfo"
ERRORINFO:
can't read "a": no such variable
    while executing
"set x $a"
    (procedure "withError" line 2)
    invoked from within
"withError 2"
```

Associative Arrays

Associative Arrays

Syntax:

```
% set name(first) "Mary"
Mary
% set name(last)  "Poppins"
Poppins
% puts "Full name: $name(first) $name(last)"
Full name: Mary Poppins
```

- `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.

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.

```
% 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
}
% global name
% set name(ID) 0
0
% addname Mary Poppins
Poppins
% addname Uriah Heep
Heep
% addname Rene Descartes
Descartes
% addname Leonardo "da Vinci"
da Vinci
% parray name
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
name(first)   = Mary
name(last)    = Poppins
% array set array1 [list {123} {Abigail Aardvark} \
                       {234} {Bob Baboon} \
                       {345} {Cathy Coyote} \
                       {456} {Daniel Dog} ]
% puts "Array1 has [array size array1] entries"
Array1 has 4 entries
% puts "Array1 has the following entries: \n [array names array1]"
Array1 has the following entries:
 345 234 123 456
% puts "ID Number 123 belongs to $array1(123)"
ID Number 123 belongs to Abigail Aardvark
% if {[array exist array1]} {
    puts "array1 is an array"
} else {
    puts "array1 is not an array"
}
array1 is an array
% if {[array exist array2]} {
    puts "array2 is an array"
} else {
    puts "array2 is not an array"
}
array2 is not an array
% for {set i 0} {$i < 5} {incr i} { set a($i) test }
% existence a(0)  ;# proc existence defined before
a(0) Exists
% unset a(0)
% existence a(0)
a(0) Does Not Exist
% existence a(3)
a(3) Exists
% existence a(4)
a(4) Exists
% catch {unset a(3) a(0) a(4)}
1
% existence a(3)
a(3) Does Not Exist
% existence a(4)
a(4) Exists
% existence a
a Exists
% array unset a *
% existence a
a Exists
% unset a
% existence a
a Does Not Exist
```

parray

parray - print an array's keys and values
Synopsis:
`parray` *arrayName ?pattern?*
Print on standard output the names and values of all the elements in the array *arrayName* that match *pattern*.

More Array Commands - Iterating and use in procedures

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

Arrays are actually collections of variables and do not have a value.

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

Instead of passing a "value" for the array, you pass the name.

```
% proc addname {db first last} {
    upvar $db name
    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} {
    # Loop over the last names: make a map from last name to ID
    upvar $db name
    foreach n [array names name "*,last"] {
        # Store in a temporary array: an "inverse" map of last name to ID
        regexp {^[^,]+} $n id
        set last       $name($n)
        set tmp($last) $id
    }
    foreach last [lsort [array names tmp]] {
        set id $tmp($last)
        puts "   $name($id,first) $name($id,last)"
    }
}
% set fictional_name(ID) 0
0
% set historical_name(ID) 0
0
% addname fictional_name Mary Poppins
Poppins
% addname fictional_name Uriah Heep
Heep
% addname fictional_name Frodo Baggins
Baggins
% addname historical_name Rene Descartes
Descartes
% addname historical_name Richard Lionheart
Lionheart
% addname historical_name Leonardo "da Vinci"
da Vinci
% addname historical_name Charles Baudelaire
Baudelaire
% addname historical_name Julius Caesar
Caesar
```

```
% puts "Fictional characters:"
Fictional characters:
% report fictional_name
   Frodo Baggins
   Uriah Heep
   Mary Poppins
% puts "Historical characters:"
Historical characters:
% report historical_name
   Charles Baudelaire
   Julius Caesar
   Rene Descartes
   Richard Lionheart
   Leonardo da Vinci
```

Dictionaries as alternative to arrays

```
% dict set clients 1 forenames Joe
1 {forenames Joe}
% dict set clients 1 surname   Schmoe
1 {forenames Joe surname Schmoe}
% dict set clients 2 forenames Anne
1 {forenames Joe surname Schmoe} 2 {forenames Anne}
% dict set clients 2 surname   Other
1 {forenames Joe surname Schmoe} 2 {forenames Anne surname Other}
% puts "Number of clients: [dict size $clients]"
Number of clients: 2
% dict for {id info} $clients {
    puts "Client $id:"
    dict with info {
       puts "   Name: $forenames $surname"
    }
}
Client 1:
   Name: Joe Schmoe
Client 2:
   Name: Anne Other
```

- `dict set` *dictionaryVariable key ?key ...? value*
- `dict unset` *dictionaryVariable key ?key ...?*
- `dict for` iterates through the items in a dictionary.
  `dict for` *keyvalnames dictionary body*
  In contrast with `foreach`, all but the last key-value pair for any redundant keys are ignored.
- `dict get` *dict ?key ...?*
- `dict with` puts dictionary values into variables named by the dictionary keys, evaluates the given script, and then reflects any changed variable values back into the dictionary.
  `dict with` *dictionaryVariable ?key ...? body*
  This command takes the dictionary and unpacks it into a set of local variables in the current scope.
- `dict update` maps values in a dictionary to variables, evaluates a script, and reflects any changes to those variables back into the dictionary.
  `dict update` *dictionaryVariable key varName ?key varName ...? body*
- `dict incr` *variable key ?increment?*
  Adds the given *increment* value (an integer that defaults to 1 if not specified) to the value that the given key maps to in the dictionary value contained in the given variable, writing the resulting dictionary value back to that variable, and returning the value of the entire dictionary. Non-existent keys are treated as if they map to 0. It is an error to increment a value for an existing key if that value is not an integer.

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.

```
% proc addname {dbVar first last} {
    upvar 1 $dbVar db
    dict incr db ID
    set id [dict get $db ID]
    dict set db $id first $first
    dict set db $id last  $last
}
% proc report {db} {
    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
    }
    foreach last [lsort [dict keys $tmp]] {
        set id [dict get $tmp $last]
```

```
        puts "   [dict get $db $id first] $last"
    }
}
% dict set fictional_name ID 0
ID 0
% dict set historical_name ID 0
ID 0 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% addname fictional_name Mary Poppins
ID 1 1 {first Mary last Poppins}
% addname fictional_name Uriah Heep
ID 2 1 {first Mary last Poppins} 2 {first Uriah last Heep}
% addname fictional_name Frodo Baggins
ID 3 1 {first Mary last Poppins} 2 {first Uriah last Heep} 3 {first Frodo last Baggins}
% addname historical_name Rene Descartes
ID 1 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% addname historical_name Richard Lionheart
ID 2 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% addname historical_name Leonardo "da Vinci"
ID 3 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% addname historical_name Charles Baudelaire
ID 4 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% addname historical_name Julius Caesar
ID 5 1 {first Rene last Descartes} 2 {first Richard last Lionheart} 3 {first Leonardo last {da Vinci}} 4 {first Charles last Baudelaire} 5 {first Julius last Caesar}
% puts "Fictional characters:"
Fictional characters:
% report $fictional_name
   Frodo Baggins
   Uriah Heep
   Mary Poppins
% puts "Historical characters:"
Historical characters:
% report $historical_name
   Charles Baudelaire
   Julius Caesar
   Rene Descartes
   Richard Lionheart
   Leonardo da Vinci
```

Modularization - source

The `source` command will load a file and execute it.
Synopsis:
`source` *filename*
`source -encoding` *encodingName 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.
`source` evaluates the contents of the specified file as a script at the level of its caller.
The first occurrence of ^Z (ASCII character 26) is interpreted as a marker indicating the end of the script. To use the character ^Z in the script, encode it as \032 or \u001a.
If *fileName* starts with a tilde (~) then `$env(HOME)` will substituted for the tilde, as is done in the `file` command.
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:

```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"
```

File Access 101

File Access

- `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
    - `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 does exist.
    - `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. The file does not exist, create it. Set the current location to the end of the file.
  - *permission* is an integer to use to set the file access permissions. 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.)
- `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`
  - `stderr`
- `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`
    - `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 of the 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` .....*Offset* is measured from the start of the file.
    - `current` ...*Offset* is measured from the current position in the file.
    - `end` .....*Offset* is measured from the end of the file.
- `tell` *fileID*
  Returns the position of the access pointer in *fileID* as a decimal string.
- `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.
- There are a finite number of open file slots available. Remember to close the files when you are done with them.
- An empty line is indistinguishable from an EOF with the command `set string [gets filename]`. Use the `eof` command to determine if the file is at the end or use the other form of `gets`.
- If you want to read "binary" data, you will have to use the `fconfigure` command.
- If you deal with configuration data for your programs, you can use the `source` command.

```
% set infile [open "myfile.txt" r]
couldn't open "myfile.txt": no such file or directory
% set number 0
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
}
can't read "infile": no such variable
% close $infile
can't read "infile": no such variable
% puts "Number of lines: $number"
Number of lines: 0
% set outfile [open "report.out" w]
file21b62434030
% puts $outfile "Number of lines: $number"
% close $outfile
```

`fconfigure` — Set and get options on a channel

Synopsis:
`fconfigure` *channelId*
`fconfigure` *channelId name*
`fconfigure` *channelId name value ?name value ...?*
The `fconfigure` command sets and retrieves options for channels.
Instruct Tcl to always send output to `stdout` immediately, whether or not it is to a terminal:

```
fconfigure stdout -buffering none
```

`-buffering` *newValue*

Information about Files - file, glob

`glob` provides the access to the names of files in a directory.
`file` provides three sets of functionality:

- String manipulation appropriate to parsing file names
  - `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
- Information about an entry in a directory:
  - `atime` ..... Returns time of last access
  - `executable` ..... Returns 1 if file is executable by user
  - `exists` ..... Returns 1 if file exists
  - `isdirectory` ..... Returns 1 if entry is a directory
  - `isfile` ..... Returns 1 if entry is a regular file
  - `lstat` ..... Returns array of file status information
  - `mtime` ..... Returns time of last data modification
  - `owned` ..... Returns 1 if file is owned by user
  - `readable` ..... Returns 1 if file is readable by 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 user
- Manipulating the files and directories themselves:

- `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

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

```
% set tclfiles [glob *.tcl]
no files matched glob pattern "*.tcl"
% puts "Name - date of last modification"
Name - date of last modification
% foreach f $tclfiles {
    puts "$f - [clock format [file mtime $f] -format %x]"
}
can't read "tclfiles": no such variable
```

- `glob` *?switches? pattern ?patternN?*
  returns a list of file names that match *pattern* or *patternN*
  - *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.
  - *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.
- `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.
- `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 ".". If the last "/" in *name* is also the first character, it 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

      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 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` *?-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 *$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*.
- `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.

```
% set dirs [glob -nocomplain -type d *]
.anaconda .android .atom .azuredatastudio .cache .comsol .conda .config .continuum .dbus-keyrings .dotnet
.InstallAnywhere .ipython .ivy2 .jupyter .kaggle .keras .matplotlib .node .omnisharp .Origin .platformio
.QtWebEngineProcess .sbt .sonarlint .SpaceVim .SpaceVim.d .ssh .standard-v14-cache .tabnine
.templateengine .vscode .wakatime .Xilinx .xp2p ansel Contacts Desktop Documents Downloads Favorites
Links Music OneDrive Pictures sangfor {Saved Games} seaborn-data Searches source Tracing Videos vimfiles
% if { [llength $dirs] > 0 } {
    puts "Directories:"
    foreach d [lsort $dirs] {
        puts "    $d"
    }
} else {
    puts "(no subdirectories)"
}
Directories:
    .InstallAnywhere
    .Origin
    .QtWebEngineProcess
    .SpaceVim
    .SpaceVim.d
    .Xilinx
    .anaconda
    .android
    .atom
    .azuredatastudio
    .cache
    .comsol
    .conda
    .config
    .continuum
```

```
    .dbus-keyrings
    .dotnet
    .ipython
    .ivy2
    .jupyter
    .kaggle
    .keras
    .matplotlib
    .node
    .omnisharp
    .platformio
    .sbt
    .sonarlint
    .ssh
    .standard-v14-cache
    .tabnine
    .templateengine
    .vscode
    .wakatime
    .xp2p
    Contacts
    Desktop
    Documents
    Downloads
    Favorites
    Links
    Music
    OneDrive
    Pictures
    Saved Games
    Searches
    Tracing
    Videos
    ansel
    sangfor
    seaborn-data
    source
    vimfiles
% set files [glob -nocomplain -type f *]
.bash_history .condarc .dotty_history .gitconfig .lesshst .notion-enhancer .npmrc .prj_config.teros
.python_history .vhdl_ls.toml .wakatime-internal.cfg .wakatime.bdb .wakatime.cfg .wakatime.data
.wakatime.db .wakatime.log .yarnrc report.out SciTE.recent SciTE.session texput.log _viminfo
% if { [llength $files] > 0 } {
    puts "Files:"
    foreach f [lsort $files] {
        puts "    [file size $f] - $f"
    }
} else {
    puts "(no files)"
}
Files:
    50 - .bash_history
    25 - .condarc
    126 - .dotty_history
    362 - .gitconfig
    20 - .lesshst
    52 - .notion-enhancer
    18 - .npmrc
    4016 - .prj_config.teros
    1410 - .python_history
    13 - .vhdl_ls.toml
    169 - .wakatime-internal.cfg
    65536 - .wakatime.bdb
    146 - .wakatime.cfg
    112 - .wakatime.data
    12288 - .wakatime.db
    234604 - .wakatime.log
    121 - .yarnrc
    0 - SciTE.recent
    21 - SciTE.session
    2911 - _viminfo
    20 - report.out
    688 - texput.log
```

Command line arguments and environment strings

Command line arguments

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`. Another method of passing information to a script is with **environment variables**. Environment variables are available to Tcl scripts in a global associative array `env`. The command `puts "$env(PATH)"` would print the contents of the `PATH` environment variable.

Environment strings

The env array is one of the magic names created by Tcl to reflect the value of the invoker's environment.
To access the env array within a Tcl proc, one needs to tell the proc that env is a global array. There are two ways to do this.

- **$::env(PATH)**
- **global env ; puts $env(PATH)**

```
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)"
}
```

info

Learning the existence of commands and variables

```
% proc safeIncr {val {amount 1}} {
    upvar $val v
    if { [info exists v] } {
        incr v $amount
    }  else {
        set v $amount
    }
}
```

- `info commands` *?pattern?*
  Returns a list of the commands, both internal commands and procedures, whose names match *pattern*.
- `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?*
  Returns a list of the local and global variables that match *pattern*.

```
% if {[info procs safeIncr] eq "safeIncr"} {
     safeIncr a
}
expected integer but got "a b c1 2 3"
% puts "After calling SafeIncr with a non existent variable: $a"
After calling SafeIncr with a non existent variable: a b c1 2 3
% set a 100
100
% safeIncr a
101
% puts "After calling SafeIncr with a variable with a value of 100: $a"
After calling SafeIncr with a variable with a value of 100: 101
% safeIncr b -3
expected integer but got "1 2 3"
% puts "After calling safeIncr with a non existent variable by -3: $b"
After calling safeIncr with a non existent variable by -3: 1 2 3
% set b 100
100
% safeIncr b -3
97
% puts "After calling safeIncr with a variable whose value is 100 by -3: $b"
After calling safeIncr with a variable whose value is 100 by -3: 97
% puts "These variables have been defined: [lsort [info vars]]"
```

```
These variables have been defined: => TMPDIR a argc argv argv0 array array1 auto_execs auto_index
auto_path b balance both change char_before clients d digits_before_period directorypath dirs e env
errorCode errorInfo examples f fictional_name files forenames fullpath hexval historical_name i id info
invert io item labels leader len line list list1 lower match mounted name number outfile outfl parens
paths pattern price1 price2 price3 price4 relativepath result sample sample2 size srtlist str2 str3
string sub1 sub2 subsetlist surname tcl_interactive tcl_library tcl_patchLevel tcl_platform
tcl_rcFileName tcl_version tempFileName trailer upper varKey1 varKey2 whole word x y
% puts "These globals have been defined:   [lsort [info globals]]"
These globals have been defined:   => TMPDIR a argc argv argv0 array array1 auto_execs auto_index
auto_path b balance both change char_before clients d digits_before_period directorypath dirs e env
errorCode errorInfo examples f fictional_name files forenames fullpath hexval historical_name i id info
invert io item labels leader len line list list1 lower match mounted name number outfile outfl parens
paths pattern price1 price2 price3 price4 relativepath result sample sample2 size srtlist str2 str3
string sub1 sub2 subsetlist surname tcl_interactive tcl_library tcl_patchLevel tcl_platform
tcl_rcFileName tcl_version tempFileName trailer upper varKey1 varKey2 whole word x y
% set exist [info procs localproc]
% if {$exist == ""} {
    puts "localproc does not exist at point 1"
}
localproc does not exist at point 1
% proc localproc {} {
    global argv
    set loc1 1
    set loc2 2
    puts "Local variables accessible in this proc are: [lsort [info locals]]"
    puts "Variables accessible from this proc are:     [lsort [info vars]]"
    puts "Global variables visible from this proc are: [lsort [info globals]]"
}
% set exist [info procs localproc]
localproc
% if {$exist != ""} {
    puts "localproc does exist at point 2"
}
localproc does exist at point 2
% localproc
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: => TMPDIR a argc argv argv0 array array1 auto_execs
auto_index auto_path b balance both change char_before clients d digits_before_period directorypath dirs
e env errorCode errorInfo examples exist f fictional_name files forenames fullpath hexval historical_name
i id info invert io item labels leader len line list list1 lower match mounted name number outfile outfl
parens paths pattern price1 price2 price3 price4 relativepath result sample sample2 size srtlist str2
str3 string sub1 sub2 subsetlist surname tcl_interactive tcl_library tcl_patchLevel tcl_platform
tcl_rcFileName tcl_version tempFileName trailer upper varKey1 varKey2 whole word x y
```

`info nameofexecutable`

Returns the full path name of the binary file from which the application was invoked. If Tcl was unable to identify the file, then an empty string is returned.

State of the interpreter

- `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.
- `info patchlevel`
  Returns the value of the global variable tcl_patchlevel. This is a three-levels version number identifying the Tcl version, like: "8.4.6"
- `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. This is the revision number of this interpreter, like: "8.4".
- `pid`
  Returns the process id of the current process.

```
% puts "This is how many commands have been executed: [info cmdcount]"
This is how many commands have been executed: 23752
% puts "Now  *THIS* many commands have been executed: [info cmdcount]"
Now  *THIS* many commands have been executed: 23761
% puts "This interpreter is revision level: [info tclversion]"
This interpreter is revision level: 8.6
% puts "This interpreter is at patch level: [info patchlevel]"
```

```
This interpreter is at patch level: 8.6.12
% puts "The process id for this program is [pid]"
The process id for this program is 2188
% 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]
23803
% set fact [factorial 3]
Current level: 1 - val: 3
Current level: 2 - val: 3
Current level: 3 - val: 3
6
% set count2 [info cmdcount]
23840
% puts "The factorial of 3 is $fact"
The factorial of 3 is 6
% puts "Before calling the factorial proc, $count1 commands had been executed"
Before calling the factorial proc, 23803 commands had been executed
% puts "After calling the factorial proc, $count2 commands had been executed"
After calling the factorial proc, 23840 commands had been executed
% puts "It took [expr $count2-$count1] commands to calculate this factorial"
It took 37 commands to calculate this factorial
% set sysdir [file dirname [info script]]
.
% source [file join $sysdir "utils.tcl"]
couldn't read file "./utils.tcl": no such file or directory
```

Information about procs

- `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 the default. Otherwise, returns 0.

```
% 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]"
The args for demo are: argument1 default
% puts "The body for demo is:  [info body demo]"
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]]"

% set arglist [info args demo]
argument1 default
% foreach arg $arglist {
    if {[info default demo $arg defaultval]} {
        puts "$arg has a default value of $defaultval"
    } else {
        puts "$arg has no default"
    }
}
argument1 has no default
default has a default value of DefaultValue
```

Running other programs from Tcl - exec, open

open

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

exec

`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* and --.
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 &`.

append

`append` appends values to the value stored in a variable.
Synopsis:
`append` *varName ?value value value ...?*
Appends each *value* to the value stored in the variable named by *varName*. If *varName* doesn't exist, it is given a value equal to the concatenation of all the *value* arguments.
`append` is a *string* command. When working with lists, definitely use `concat` or `lappend`.

```
% set a [list a b c]
a b c
% set b [list 1 2 3]
1 2 3
% append a $b
a b c1 2 3
% puts $a ;# -> a b c1 2 3
a b c1 2 3
```

To assign the empty string to a variable if it doesn't already exist, leaving the current value alone otherwise: `append var {}`.

```
% set TMPDIR "/tmp"
/tmp
% if { [info exists ::env(TMP)] } {
    set TMPDIR $::env(TMP)
}
C:\Users\Yihua\AppData\Local\Temp
% set tempFileName "$TMPDIR/invert_[pid].tcl"
C:\Users\Yihua\AppData\Local\Temp/invert_2188.tcl
% set outfl [open $tempFileName w]
file21b63da0550
% puts $outfl {
    set len [gets stdin line]
    if {$len < 5} {exit -1}
    for {set i [expr {$len-1}]} {$i >= 0} {incr i -1} {
        append l2 [string range $line $i $i]
    }
    puts $l2
    exit 0
}
% flush $outfl
% close $outfl
% set io [open "|[info nameofexecutable] $tempFileName" r+]
file21b63dc9c70
% # send a string to the new program     *MUST FLUSH*
% puts $io "This will come back backwards."
% flush $io
% set len [gets $io line]
-1
% puts "Reversed is: $line"
Reversed is:
% puts "The line is $len characters long"
The line is -1 characters long
% set invert [exec [info nameofexecutable] $tempFileName << \
       "ABLE WAS I ERE I SAW ELBA"]
ABLE WAS I ERE I SAW ELBA
% puts "The inversion of 'ABLE WAS I ERE I SAW ELBA' is \n $invert"
The inversion of 'ABLE WAS I ERE I SAW ELBA' is
 ABLE WAS I ERE I SAW ELBA
% file delete $tempFileName
```

Building reusable libraries - packages and namespaces

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.

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.
- Installing the package where it can be found by Tcl.

Commands:

- `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, otherwise, 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 any version will be loaded.
- `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 is able to handle both Tcl script files and binary libraries.

Namespaces

- `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

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.

```
# 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
}
```

```
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
```

`variable`

Synopsis:
`variable` *?name value...? name ?value?*
Declares a variable named *name*, relative to the current namespace, and binds that variable to the `tail` of that name in the current evaluation level. If a corresponding value is provided, the variable is `set` to that value.

```
% namespace eval one {
    variable greeting hello
}
% set one::greeting ;#-> hello
hello
```

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.
`namespace ensemble` creates and configures namespace ensembles.

```
namespace eval glovar {
    namespace export getit setit
    namespace ensemble create
    variable value {};
    proc getit {} {
        variable value
        return $value
    }
    proc setit newvalue {
        variable value
        set value $newvalue
    }
}
```

Three subcommands of namespace ensemble are:

- `namespace ensemble create` *?option value ...?*
  Creates a new ensemble linked to the current namespace and returns the fully-qualified name of the new ensemble.
- `namespace ensemble configure` *ensemble ?option? ?value ...?*
  Retrieves the value of an option associated with *ensemble*, or configures some options associated with that ensemble.

Returns true if invoking *ensemble* would invoke a namespace ensemble command, and false otherwise.

```
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
```

Creating Commands - eval

The **eval** command will evaluate a list of strings as though they were commands typed at the **%** prompt or sourced from a file.
Synopsis:
`eval` *arg1 ??arg2?? ... ??argn??*
Evaluates *arg1 - argn* as one or more Tcl commands. The *args* are concatenated into a string, and passed to **tcl_Eval** to evaluate and execute.

```
set cmd {puts "Evaluating a puts"}
puts "CMD IS: $cmd"
eval $cmd
if {[string match [info procs newProcA] ""] } {
    puts "Defining 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 "The body of newProcA is: \n[info body newProcA]"
puts "newProcA returns: [newProcA]"
puts "newProcA returns: [newProcA]"
# Define a proc using lists
if {[string match [info procs newProcB] ""] } {
    puts "Defining newProcB for this invocation"
    set cmd "proc newProcB "
    lappend cmd {}
    lappend cmd {global num; incr num; return $num;}
    eval $cmd
}
puts "The body of newProcB is: \n[info body newProcB]"
puts "newProcB returns: [newProcB]"
```

Output:

> CMD IS: puts "Evaluating a puts"
> CMD IS: puts "Evaluating a puts"
> Evaluating a puts
> The body of newProcA is:
> global num; incr num; return "/tmp/TMP.2188.
> $num"; newProcAreturns : /tmp/TMP.2188.7newProcAreturns : /tmp/TMP.2188.8Thebodyof newProcBis : globalnum; incrnum
> num;
> newProcB returns: 9

More command construction - format, list

```
% eval [list puts {Not OK}]
Not OK
% eval [list puts "Not OK"]
Not OK
% set cmd "puts" ; lappend cmd {Not OK}; eval $cmd
Not OK
```

`info complete` *string*
If *string* has no unmatched brackets, braces or parentheses, then a value of 1 is returned, else 0 is returned.

```
% set cmd "NOT OK"
```

```
NOT OK
% eval puts $cmd
can not find channel named "NOT"
% eval [format {%s "%s"} puts "Even This Works"]
Even This Works
% set cmd "And even this can be made to work"
And even this can be made to work
% eval [format {%s "%s"} puts $cmd ]
And even this can be made to work
% set tmpFileNum 0;
0
% set cmd {proc tempFileName }
proc tempFileName
% lappend cmd ""
proc tempFileName {}
% lappend cmd "global num; incr num; return \"/tmp/TMP.[pid].\$num\""
proc tempFileName {} {global num; incr num; return "/tmp/TMP.2188.$num"}
% eval $cmd
% puts "This is the body of the proc definition:"
This is the body of the proc definition:
% puts "[info body tempFileName]"
global num; incr num; return "/tmp/TMP.2188.$num"
% set cmd {puts "This is Cool!}
puts "This is Cool!
% if {[info complete $cmd]} {
    eval $cmd
} else {
    puts "INCOMPLETE COMMAND: $cmd"
}
INCOMPLETE COMMAND: puts "This is Cool!
```

Substitution without evaluation - format, subst

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.
Synopsis:
`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.

```
% set a "alpha"
alpha
% set b a
a
% puts {a and b with no substitution: $a $$b}
a and b with no substitution: $a $$b
% puts "a and b with one pass of substitution: $a $$b"
a and b with one pass of substitution: alpha $a
% puts "a and b with subst in braces: [subst {$a $$b}]"
a and b with subst in braces: alpha $a
% puts "a and b with subst in quotes: [subst "$a $$b"]"
a and b with subst in quotes: alpha alpha
% puts "format with no subst [format {$%s} $b]"
format with no subst $a
% puts "format with subst: [subst [format {$%s} $b]]"
format with subst: alpha
% eval "puts \"eval after format: [format {$%s} $b]\""
eval after format: alpha
% set num 0;
0
% set cmd "proc tempFileName {} "
proc tempFileName {}
% set cmd [format "%s {global num; incr num;" $cmd]
proc tempFileName {}  {global num; incr num;
% set cmd [format {%s return "/tmp/TMP.%s.$num"} $cmd [pid] ]
proc tempFileName {}  {global num; incr num; return "/tmp/TMP.2188.$num"
% set cmd [format "%s }" $cmd ]
proc tempFileName {}  {global num; incr num; return "/tmp/TMP.2188.$num" }
% eval $cmd
% puts "[info body tempFileName]"
global num; incr num; return "/tmp/TMP.2188.$num"
% set a arrayname
arrayname
% set b index
index
% set c newvalue
```

```
newvalue
% eval [format "set %s(%s) %s" $a $b $c]
newvalue
% puts "Index: $b of $a was set to: $arrayname(index)"
Index: index of arrayname was set to: newvalue
```

Changing Working Directory - cd, pwd

- `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 users 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.

```
% set dirs [list TEMPDIR]
TEMPDIR
% puts "[format "%-15s  %-20s " "FILE" "DIRECTORY"]"
FILE             DIRECTORY
% foreach dir $dirs {
    catch {cd $dir}
    set c_files [glob -nocomplain c*]
    foreach name $c_files {
        puts "[format "%-15s  %-20s " $name [pwd]]"
    }
}
Contacts         C:/Users/Yihua
```

More Debugging - trace

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*

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.

To set a trace on a variable so that when it's written to, the value doesn't change:

```
% proc vartrace {oldval varname element op} {
    upvar $varname localvar
    set localvar $oldval
}
% set tracedvar 1
1
% trace add variable tracedvar write [list vartrace $tracedvar]
% set tracedvar 2
1
% puts "tracedvar is $tracedvar"
tracedvar is 1
```

```
% 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"
TRACE: Write i2 in main
testvalue
% puts "call testProc"
call testProc
% testProc i1 i2
TRACE: Write i in testProc i1 i2
TRACE: Read j in testProc i1 i2
testvalue
% puts "Traces on i1: [trace info variable i1]"
Traces on i1: {write traceproc}
% puts "Traces on i2: [trace info variable i2]"
Traces on i2: {write traceproc} {read traceproc}
% trace remove variable i2 read traceproc
% puts "Traces on i2 after vdelete: [trace info variable i2]"
Traces on i2 after vdelete: {write traceproc}
% puts "call testProc again"
call testProc again
% testProc i1 i2
TRACE: Write i in testProc i1 i2
testvalue
```

Timing scripts

`time` will measure the length of time that it takes to execute a script.
Synopsis:
`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.

```
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} 10]"
puts "Time for array index: [ time {timetst2 array} 10]"
```

Output:

> Time for list search: 86.88000000000001 microseconds per iteration
> Time for array index: 3.6 microseconds per iteration

After you've run the example, play with the size of the loop counters in `timetst1` and `timetst2`. If you make the inner loop counter 5 or less, it may take longer to execute `timetst2` than it takes for `timetst1`. This is because it takes time to calculate and assign the variable `k`, and if the inner loop is too small, then the gain in not doing the multiply inside the loop is lost in the time it takes to do the outside the loop calculation.

Channel I/O: socket, fileevent, vwait

`socket`

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.

- `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`

- `fileevent` *channelID readable ?script?*
- `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`

- `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.

```
proc serverOpen {channel addr port} {
    global connected
    set connected 1
    fileevent $channel readable "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
    }
}
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
```

`after`

Synopsis:

- `after` *ms*
- `after` *ms script ?script script ...?*
- `after cancel` *id*
- `after cancel` *script script script ...*
- `after idle` *?script script script ...?*
- `after info` *?id?*

`after` uses the system time to determine when it is time to perform a scheduled event. This means that with the exception of `after 0` and `after idle`, it can be subverted by changes in the system time.
`after idle` schedules a script for evaluation when there are no other events to process.
`after cancel` the scheduled script corresponding to *id*.
`after info` provides information about the corresponding *id*, or about all scheduled scripts.
Asynchronous Mode Example

```
proc sync {} {
    after 1000
    puts {message 1}
    puts {message 2}
}
proc async {} {
    after 1000 [list puts {message 1}]
    puts {message 2}
}
```

`update`

Synopsis:
`update` *?idletasks?*
update services all outstanding events, including those that come due while `update` is operating. `update` works by performing the following steps in a loop until no events are serviced in one iteration:

1. Service the first event whose scheduled time has come.
2. If no such events are found, service all events currently in the idle queue, but not those added once this step starts.

Time and Date - clock

The **clock** command provides access to the time and date functions in Tcl.

- `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*.
  - 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.
  - The **-format** option controls what format the return will be in.
- `clock scan` *dateString -option value...?*
  The **scan** subcommand converts a human readable string to a system clock value, as would be returned by **clock seconds**. The `-format` option is used to describe the format of the *dateString*. If **-format** is not used, the command tries to guess the format of *dateString*.

```
% set systemTime [clock seconds]
1660580120
% puts "The time is: [clock format $systemTime -format %H:%M:%S]"
The time is: 00:15:20
% puts "The date is: [clock format $systemTime -format %D]"
The date is: 08/16/2022
% puts [clock format $systemTime -format {Today is: %A, the %d of %B, %Y}]
Today is: Tuesday, the 16 of August, 2022
% puts "the default format for the time is: [clock format $systemTime]"
the default format for the time is: Tue Aug 16 00:15:20 CST 2022
% set halBirthBook "Jan 12, 1997"
Jan 12, 1997
% set halBirthMovie "Jan 12, 1992"
Jan 12, 1992
% set bookSeconds [clock scan $halBirthBook -format {%b %d, %Y}]
852998400
% set movieSeconds [set movieSeconds [clock scan $halBirthMovie -format {%b %d, %Y}]]
695145600
% puts "The book and movie versions of '2001, A Space Oddysey' had a"
The book and movie versions of '2001, A Space Oddysey' had a
% puts "discrepancy of [expr {$bookSeconds - $movieSeconds}] seconds in how"
discrepancy of 157852800 seconds in how
% puts "soon we would have sentient computers like the HAL 9000"
soon we would have sentient computers like the HAL 9000
```

More channel I/O - fblocked & fconfigure

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.
  - **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 Macintosh 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.

```
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 ]"
    # 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;
}
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"
# 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}
```

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.

Child interpreters

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.
The primary interpreter (what you get when you type tclsh) is the empty list `{}`.

- `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` *args*
  This is similar to the regular `eval` command, except that it evaluates the script in the child interpreter 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 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.
  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 was 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.

```
set i1 [interp create firstChild]
set i2 [interp create secondChild]
puts "first child interp:  $i1"
puts "second child interp: $i2"
# 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
# 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]"
```

Output:

```
TRACE: Write i1 in main
TRACE: Write i2 in main
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
firstChild reports rtnName is: Contacts
```