TCL Cheat Sheet

Print to console

```
puts "Hello, World!" ;# prints text in double quotes
puts {Hello, World!} ;# prints text in curly braces

puts -nonewline Hello
puts World! ;# prints text without quotes or braces
```

Comments

Comments in TCL are either at the beginning of a line and starting with a `#` or in between a line and starting with `;#`. The semicolon is important to separate TCL commands from the comment.

```
# This is a comment
puts "Hello, World!" ;# This is also a comment
```

Variables

Variables in TCL are defined using the `set` command. The content of the variable can be accessed using the `$` sign.

```
set name "John Doe"
puts $name
```

The value of a variable can also be accessed within a string:

```
set name "John Doe"
puts "Hello, $name" ;# prints "Hello, John Doe"
```

To forget the content of a variable, the `unset` command can be used.

```
set name "John Doe"
puts $name ;# prints "John Doe"
unset name
puts $name ;# Error: can't read "name": no such variable
```

Escaped Sequences

Using the `$` sign, the value of a variable can be substituted in a string. What do we do when we want to print the raw text instead of performing a substitution? For this special case, the `$` sign can be escaped using the backslash `\`.

```
set name "John Doe"
puts "Hello, $name" ;# prints "Hello, John Doe"
puts "Hello, \$name" ;# prints "Hello, $name"
```

There are also other special characters that can be escaped using the backslash `\`: - `\n` - newline - `\t` - tab - `\"` - double quote - `\\` - backslash

```
puts "This string is two li\nes long."
puts "This string has a \tab in the middle."
puts "This string has a \"quote\" in it."
```

The backslash can be used to escape the last character of a line to continue the command on the next line.

```
puts "This is a very long string \
that spans multiple lines."
```

It is highly recommended to use the backslash to escape a dollar sign when it is not used for variable substitution, even though most of the times it also seems to work without escaping it.

```
set a anything
puts "$$a is good but \$$a is better."
```

Curly Brackets

Curly brackets `{}` can be used in a similar way as double quotes `""` to define a string. The big difference between the two is that the curly brackets do not perform variable substitution. Even the backslash does not escape the special characters in curly braces as it does in double quotes.

```
set phrase "The lemon is sour"
puts "$phrase" ;# prints "The lemon is sour"
puts {$phrase} ;# prints "$phrase"
# further examples
puts "{$phrase}" ;# prints the phrase with curly braces
puts {I don't think, "the lemon tastes sour."} ;# prints the phrase, including the double q
```

The only thing that is substituted in curly braces is the backslash \ to continue the command on the next line.

```
puts {This is a very long string\
that spans multiple lines.}
# prints "This is a very long string that spans multiple lines."
```

In this example the backslash in combination with the line break is replaced with a space.

Square Brackets: Interpolation

Commands within square brackets are evaluated before the surrounding command is executed. The result of the command within the square brackets is substituted in the surrounding command which is executed afterwards. This process is called interpolation. At first a simple example shows hot two variables can be initialised with the same value, using the `set` command which returns the value of the variable it sets.

```
set a [set b 42]
puts $a ;# prints 42
puts $b ;# prints 42
```

In this example the `set` command within the square brackets is executed first and returns the value `42`. This value is then substituted in the surrounding `set` command which sets the variable `a` to `42`. This substitution only works when it is not being used in curly braces.

```
set z {[set x "string 1"]}
puts $z ;# prints [set x "string 1"]
puts $x ;# Error: can't read "x": no such variable
```

Double quotes allow for a substitution of the command within a string.

```
set z "[set x "string 1"]"
puts $z ;# prints string 1
puts $x ;# prints the same thing
```

Within double quotes the square brackets can be escaped using the backslash.

```
set z "\[set x "string 1"\]"
puts $z ;# prints [set x {string 1}]
puts $x ;# Error: can't read "x": no such variable
```

Arithmetic Operators

The `expr` command is used to evaluate arithmetic expressions and return the result as a string. The result can be stored in a variable using interpolation:

```
set a 5
set b 3
puts [expr $a + $b] ;# prints 8
puts [expr {$a + $b}] ;# prints 8
```

Performance Tipp: Using curly braces `{}` around the expression (`{$a + $b}`) is faster than using no braces (`$a + $b`) because then the result is just the string which can be evaluated later. This might come at the cost of some side effects.

The `expr` command supports a variety of arithmetic operators as well as mathematical functions (`sin()`, `cos()`, `log()` which is the natural logarithm to the base e, `sqrt()` and many others) and the evaluation of boolean expressions. Within the expression brackets are used the same way as in regular mathematical expressions. Many commands use `expr` behind the scene, such as `if`, `while` and `for` which are discussed later.

If there are two operators applied to the same operand, the operators are applied along a pre defined preference order. The following example demonstrates this behavior with the addition and the multiplication operator. It also shows that brackets change the order of the evaluation.

```
puts [expr {2 + 2 * 4}] ;# prints 10
puts [expr {(2 + 2) * 4}] ;# prints 16
```

Operands

Where possible, operands are interpreted as integer values. They might be specified in decimal, in binary with the first two characters being `0b`, in octal with the first character being `0o` or in hexadecimal with the first two characters being `0x`. For compatibility reasons with older TCL versions an octal value is also detected when only the first character is `0`. The following example demonstrates this behavior where each statements prints the number `10`.

```
puts [expr 10]
puts [expr 0b1010]
puts [expr 0o12]
puts [expr 0xA]
```

If no integer representation of the operand is possible, the operand is interpreted as a floating point number. There are many ways to specify a floating point number, as shown in the following example:

```
puts [expr 3.14]
puts [expr 3.] ;# the decimal part is 0
puts [expr .14] ;# the integer part is 0
puts [expr 3.5e-1] ;# scientific notation
```

If none of these representations is possible, the operand is interpreted as a string.

```
set number 3,14 ;# the comma is not the decimal separator!
expr {$number > 3} ;# True, because the string "3,14" alphabetically comes after "3"
expr {$number > 3.0} ;# False, because the string "3,14" alphabetically comes before "3.0"
```

Operators for integers

The following table lists all the operators that work on numbers (integers and floats). This table is sorted by the hierarchy of the operators: Operators listed first in this list are the preferred ones.

Operator	Description	Explaination
- +	unary minus, unary plus	multiplication with (-1) / with (+1)
**	exponent	
* / %	multiplication, division, modulo	modulo is the remainder of a division
+ -	addition, subtraction	
« »	shift left / shift right	perform a bitwise shift
< <= > >=	comparison	less, less or equal, greater, greater or equal

Operator	Description	Explaination
== !=	equality, inequality	
~ !	bitwise NOT, logical NOT	
&	bitwise AND	
ˆ	bitwise XOR	
|	bitwise OR	
&&	logical AND	
||	logical OR	
? :	conditional (ternary operator)	later explained in detail

Operators for strings

There are only two groups of operators available to compare strings with each other:

Operator	Description	Explaination
== != < > <= >=	comparison	less, less or equal, greater, greater or equal
eq ne lt gt le ge	comparison	less, less or equal, greater, greater or equal

Ternary operator

The ternary operator `? :` is a conditional operator that is used to evaluate a boolean expression. It can be described as a short form of an `if - else` statement: `condition ? return-if-true : return-if-false`. The following example demonstrates the usage of the ternary operator.

```
set x 1
expr{ $x % 2 ? "Odd" : "Even" } ;# prints "Odd"
```

In this example the condition `1 % 2` is evaluated to `1` which is interpreted as `True` (everything which is not equal to zero is treated as `True`). The string `"Odd"` is returned if the condition is `True` and the string `"Even"` is returned if the condition is `False`.

Mathematical functions

By default there is a set of mathematical functions available in TCL. These functions can be used in the `expr` command. The following example demonstrates the usage of the `sqrt()` function.

```
puts [expr {sqrt(9)}] ;# prints 3.0
```

The following table lists all the mathematical functions that are available in TCL:

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	

Data types

TCL does not know many data types beside strings. There are four different representations of number: `double`, `int` (integer), `wide` (large integer) and `entire` (integer or wide, depending on the context). These are also the names of the functions to convert between the data types.

- `double` contains a (double precision) floating point number
- `int` contains an integer value. The conversion from a double to an integer truncates the decimal part.
- `wide` contains a large integer value. It behaves the same as `int` but can store larger numbers.
- `entire` contains an integer of appropriate size. Depending on the number itself the data type is either `int` or `wide` or an integer of arbitrary size.

These representations are important because computers internally have different methods to store numbers. There are some rules for the data type of the result of an operation: If both operands are integers, the result is an integer. If at least one of the operands is a double, the result is a double. This can lead to some unexpected results:

```
puts [expr 1/2] ;# prints 0 because the decimal part is truncated
puts [expr -1/2] ;# prints -1
puts [expr 1.0/2] ;# prints 0.5
puts [expr 1/2.0] ;# prints 0.5
puts [expr double(1)/2] ;# prints 0.5
```

The representation for floating point values is not always exact because there are some numerical abbreviations. These are regularly really small but can ad up and lead to some unexpected results.

```
set pi1 [expr {4*atan(1)}]
set pi2 [expr {6*asin(0.5)}]
puts [expr {$pi1-$pi2}] ;# -4.440892098500626e-16
```

Examples

Square root

```
set x 100
set y 256
puts "The square root of [expr $x + $y] is [expr {sqrt($x + $y)}]"
```

Hypotenuse of a triangle

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

Attention: Numbers with leading zero

Every number that starts with a `0` is interpreted as an octal number. Therefore the number `0700` is interpreted as `7*8*8 = 448`. This produces errors, when the number contains an `8` or a `9`.

```
expr {0900+1} ;# should be "$0900" or "{0900}" or "0900(...)" or ... (invalid octal number?
```

Control Structures: If

The syntax for the `if` statement is as follows, where the `else` and `elseif` parts are optional (indicated with question marks):

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

There are different ways how a condition can be evaluated:

	False	True
numeric value	0	everything else
string “Yes” / “No”	“No”	“Yes”
True / False	False	True

Strings which are identical to “Yes”, “No”, “True” or “False” are interpreted as boolean values. There is no case sensitivity, meaning that “YeS” and “nO” are also valid boolean values.

If the condition is true then the body of the `if` statement is executed. Otherwise the next condition is checked and if the last condition is false the `else` part is executed.

```
set x 1

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

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

In these examples it is important that an opening curly brace `{` is on the same line as the previous closing curly brace `}` and the condition is in the same line as the `if` statement. Otherwise the `if` statement is not recognized as a single command.

Control Structures: Switch

Instead of a long chain of `if - elseif - else` statements, the `switch` statement can be used. The syntax is as follows:

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

The `switch` statement compares the string with the patterns. If the first pattern matches the string, the body of the first pattern is executed. If the first pattern does not match the string, the next pattern is checked. This process is repeated until a pattern matches the string or the end of the `switch` statement is reached.

If none of the patterns match the string and a `default` pattern is provided, the body of the `default` pattern is executed. This pattern is optional and needs to be placed as the last pattern in the `switch` statement. This guarantees that some set of code will be executed no matter what the content of the string is. If there is no `default` pattern and none of the patterns match the string, the `switch` command returns an empty string.

By default the `pattern` uses the *glob-style matching* where the asterisk `*` matches any sequence of characters, e.g, the pattern `chocolate*` matches the strings `chocolate`, `chocolates` or `chocolate-cake` but not `choco-late`. This can be disabled by using `-exact` for the options, causing the patterns to be treated as strings which have to be matched exactly. Another alternative is to use `-regexp` for the options, which allows for regular expressions to be used as patterns, which are described later in detail.

Here are some examples for the use of the `switch` statement:

```
set x 1
switch $x {
    1 {puts "One"}
    2 {puts "Two"}
    3 {puts "Three"}
    default {puts "Something else"}
}
```

When there are two patterns which want to use the same body, the `-` sign tells to use the body of the next pattern. The following example demonstrates this behavior:

```
set str "a"
switch -exact $str {
    "a" -
    "b" {
        puts "Using the body of pattern 'b' for both a and b"
    }
}
```

The `switch` statement becomes more useful than nested `if - elseif - else` statements when there are many conditions to check:

```
set edges 3

switch $edges {
    0 -
    1 -
    2 {puts "Not a polygon"}
    3 {puts "Triangle"}
    4 {puts "Quadrilateral"}
    5 {puts "Pentagon"}
    6 {puts "Hexagon"}
    # further patterns ...
    default {puts "Unknown polygon"}
}
```

The last example shows that variable substitution is not available for the patterns because the whole `switch` statement is surrounded by curly braces:

```
set x [set y 1] ;# set both variables to 1

switch $x {
   $y {puts "The value of x is the same as the value of y"}
   1 {puts "The value of x is 1"}
   default {puts "No match for x"}
}
```

This example prints the text *The value of x is 1.* When the variable x is set the following way: `set x {$y}` the text *The value of x is the same as the value of y* is printed.

Alternative Syntax

Regularly the patterns do not allow for a variable substitution because the whole `switch` statement is surrounded by curly braces. To bypass this limitation the `switch` statement supports an alternative syntax in which the curly braces are omitted and all the patterns are passed to the switch statement as arguments:

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

In this case the backslashes are necessary to continue the switch command on the next line. The following example demonstrates the usage of the alternative syntax for the previous example:

```
set x [set y 1]

switch $x\
    $y {
        puts "The value of x is the same as the value of y"
    }\
    1 {
        puts "The value of x is 1"
    }\
    default {
        puts "No match for x"
    }
# prints "The value of x is the same as the value of y"
```

Control Structures: While loop

//TODO