Tcl Variable Utilities

Version 2.1

Alex Baker

https://github.com/ambaker1/vutil

October 3, 2023

Abstract

This package provides various utilities for working with variables in Tcl, including read-only variables, TclOO garbage collection, and an object-variable type system.

Default Values

The command *default* assigns values to variables if they do not exist.

```
default $varName $value
```

$varName	Name of variable to set
$value	Default value for variable

The example below shows how default values are only applied if the variable does not exist.

Example 1: Variable defaults

Code:

```
set a 5
default a 7; # equivalent to "if {![info exists a]} {set a 7}"
puts $a
unset a
default a 7
puts $a
```

Output:

```
5
7
```

Variable Locks

The command *lock* uses Tcl variable traces to make a read-only variable. If attempting to modify a locked variable, it will throw a warning, but not an error.

```
lock $varName <$value>
```

$varName	Variable name to lock.
$value	Value to lock variable at. Default self-locks (uses current value).

The command *unlock* unlocks previously locked variables so that they can be modified again.

```
unlock $name1 $name2 ...
```

$name1 $name2 ...	Variables to unlock.

Example 2: Variable locks

Code:

```
lock a 5
set a 7; # throws warning to stderr channel
puts $a
unlock a
set a 7
puts $a
```

Output:

```
failed to modify "a": read-only
5
7
```

Note: You can lock array elements, but not an entire array.

Variable-Object Ties

As of Tcl version 8.6, there is no garbage collection for Tcl objects, they have to be removed manually with the "destroy" method. The command *tie* is a solution for this problem, using variable traces to destroy the corresponding object when the variable is unset or modified. For example, if an object is tied to a local procedure variable, the object will be destroyed when the procedure returns.

```
tie $refName <$object>
```

$refName	Name of reference variable for garbage collection.
$object	Object to tie variable to. Default self-ties (uses current value).

In similar fashion to *unlock*, tied variables can be untied with the command *untie*.

```
untie $name1 $name2 ...
```

$name1 $name2 ...	Variables to untie.

Example 3: Variable-object ties

Code:

```
oo::class create foo {
    method hi {} {
        puts hi
    }
}
tie a [foo create bar]
set b $a; # alias variable
unset a; # triggers ``destroy''
$b hi; # throws error
```

Output:

```
invalid command name "::bar"
```

Note: You can tie array elements, but not an entire array, and you cannot tie a locked variable.

Reference Variables

Valid reference variables for the *tie* command must match the following regular expression:

(::+|\w+)+(\(\w+\))?

The one exception to this rule is the shared global reference variable "**&**". This shared reference, regardless of scope, can be accessed with the command "**$&**".

```
$& $arg ...
```

$arg ...	Arguments for object.

Reference Parser API

The command *::vutil::RefSub*, which performs "**$@ref**" substitution on a given string, returning the updated string and all matched reference names. For example, "**$@ref**" is converted to "**${::@(ref)}**": an element of the global array variable "**@**". To escape a reference, especially for nested substitution, simply add more "**@**" symbols, like "**$@@ref**". This command is not exported, as it is intended for use by developers.

```
::vutil::RefSub $string
```

$string	String to perform substitution with.

There are two special references: "**$@&**" and "**$@.**". Both refer to the global variables "**&**" and "**.**", respectively, and they are always listed first in the reference variable list, as shown in the example below:

Example 4: Reference variable substitution

Code:

```
lassign [::vutil::RefSub {$@& + $@x(1) - $@@y + $@.}] string refs
puts $string
puts $refs
```

Output:

```
${::@(::&)} + ${::@(x(1))} - $@y + ${::@(::.)}
::& ::. x(1)
```

Garbage Collection Superclass

The class "`::vutil::GC`" is a TclOO superclass that includes garbage collection. This class is not exported, and not intended for direct use, as it is simply a template for classes with built-in garbage collection, by tying the object to a specified reference variable using *tie*. In addition to tying the object to a reference variable in the superclass constructor, the "`::vutil::GC`" superclass also provides a method for copying the object to a new reference variable: "`-->`".

```
$obj --> $refName
```

`$obj`	Object that inherits the "`::vutil::GC`" superclass.
`$refName`	Name of reference variable for garbage collection.

Below is an example of how this superclass can be used to build garbage collection into a TclOO class.

Example 5: Creating a class with garbage collection

Code:

```
oo::class create container {
    superclass ::vutil::GC
    variable myValue
    constructor {refName value} {
        set myValue $value
        next $refName
    }
    method set {value} {set myValue $value}
    method value {} {return $myValue}
}
proc wrap {value} {
    container new & $value
    return $&
}
[wrap {hello world}] --> x
puts [$x value]
unset x; # also destroys object
```

Output:

```
hello world
```

Object Variable Class

The TclOO class *var* is a subclass of "`::vutil::GC`" that acts as a container class for data, so that calling the object variable by itself (e.g. `[$varObj]`) returns the value stored in the object.

```
var new $refName <$value>
```

`$refName`	Name of reference variable for garbage collection.
`$value`	Value to set object variable to. Default blank.

Standard Object Variable Operators

The operator method "`=`" assigns the value of the object variable, returning the name of the object. The operator method "`<-`" assigns the value and any object metadata directly from another object variable of the same class, returning the name of the object. The operator method "`-->`" copies the object variable to a new reference variable, returning the name of the new object.

```
$varObj = $value
```

```
$varObj <- $otherVarObj
```

```
$varObj --> $refName
```

`$value`	Value to set object variable to.
`$otherVarObj`	Other object variable to assign value from (must be same class).
`$refName`	Name of reference variable for garbage collection.

The example below demonstrates the myriad of ways object variables can be manipulated:

Example 6: Standard object variable operators

Code:

```
var new x; # Create blank object variable $x
[$x --> y] = 2; # Copy $x to $y, and set to 2
[var new z] <- [$x <- $y]; # Create $z and set to $x after setting $x to $y.
puts [list [$x] [$y] [$z]]
```

Output:

```
2 2 2
```

Advanced Object Variable Operators

The operator methods "`.=`", "*`:=`*", and "*`::=`*" are similar to the operator method "`=`", except that they can perform a transformation on the existing value of the object variable.

```
$varObj .= $oper
```

```
$varObj := $expr
```

```
$varObj ::= $body
```

`$oper`	Tcl math operator and additional mathop arguments, i.e. `"+ 5"`.
`$expr`	Tcl expression to evaluate, passing input through the Tcl *expr* command.
`$body`	Tcl script to evaluate.

The operator "`.=`" passes the value of the object as the first argument in the corresponding Tcl mathop command, allowing for modification in reference to the current value. Similarly, the operators "*`:=`*" and "*`::=`*", allow for object self-reference using the command "*$.*", which accesses the global read-only self-reference variable "`.`".

```
$.  <$arg ...>
```

`$arg ...`	Arguments for object.

The example below demonstrates how you can use these features to manipulate object variables.

Example 7: Advanced object variable operators

Code:

```
var new x 5.0; # Create variable $x
[[var new y] <- $x] .= {+ 10}; # Create new variable y, set to x, and add 10.
set p 2; # Create primative variable
$y := {[$.] ** $p + [$x]}; # Square y, plus $x (230.0) (accesses $p)
$y ::= {split [$.] .}; # Split at decimal (230 0)
puts [$y]
```

Output:

```
230 0
```

Object Variable Metadata

The method *info* accesses all object variable metadata. Fields "type" and "value" are standard.

```
$varObj info <$field>
```

$field	Info field to get. Default returns dictionary of all info.

Printing an Object Variable Value

The method *print* prints the value of the variable to screen or file.

```
$varObj print <-nonewline> <$channelID>
```

-nonewline	Option to print without newline.
$channelID	Channel ID open for writing. Default stdout.

Destroying an Object Variable

Because object variables are simply TclOO objects, they can be destroyed with the standard method *destroy*. Additionally, unsetting the linked variable will also destroy the object.

```
$varObj destroy
unset $refName
```

$refName	Name of reference variable used for garbage collection.

Example 8: Standard object variable methods

Code:

```
var new x {Hello World}
puts [$x info]
$x print
$x destroy; # or "unset x"
```

Output:

```
type var value {Hello World}
Hello World
```

Object Variable Types

The TclOO class *var* acts as a superclass for a pure-Tcl type system. Type classes are created and managed through the command ensemble *type*.

```
type $subcommand $arg ...
```

$subcommand	Subcommand name.
$arg ...	Arguments for subcommand.

New type classes can be created using the subcommands *new* or *create*. Both subcommands create a class that is a subclass of "`::vutil::var`", with a private method *Type* that returns the corresponding type. If creating a type class with the subcommand *new*, the resulting class will be named "`::vutil::type.$type`".

```
type new $type $defScript
type create $type $name $defScript
```

$type	Name of type.
$name	Name of class.
$defScript	Class definition script.

Note: The value of the object variable is stored in the blank array name "value". The blank array is used to store all the object variable properties, and is what is returned with the method "info". By default, from the superclass "`::vutil::var`", there are two properties: (value) and (type).

Type Queries

A list of all defined types can be queried with the subcommand *names*.

```
type names
```

The existence of a type can be queried with the subcommand *exists*, and the class associated with a type can be queried with the subcommand *class*.

```
type exists $type
```

```
type class $type
```

$type	Name of type.

The subcommand *isa* checks if an object is of a specific type or of one of its subtypes. If the type or object does not exist, this command will return an error. Similarly, the subcommand *assert* returns an error if an object is not of a specific type or of one its subtypes.

```
type isa $type $object
```

```
type assert $type $object
```

$type	Name of type.
$object	Name of object.

Basic Type Library

This package provides a few basic object variable types: *var*, *string*, *bool*, *int* and *float*.

Creating Type Variables

Classes defined by *type* only have the constructor method *new*, so as a convenience, the command *new* creates a new variable object of a specified type. If the reference name provided is blank, it will simply return the value after passing it through the datatype's data validation.

```
new $type $refName <$value>
```

$type	Name of type.
$refName	Name of reference variable to tie to object. Blank to return value.
$value	Value to set object variable to (default varies).

Type "var" (object variable)

The type "*var*" is just an alternative way to create an object variable (same syntax as "new" method for *var* class). It is the superclass for all other types.

```
new var $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	Value to set object variable to (default blank).

Example 9: Basic object variable

Code:

```
new var a
puts [$a info]
[$a = foobar] print
```

Output:

```
type var value {}
foobar
```

Type "string"

The type "*string*" does not do any validation on input (because in Tcl, "everything is a string"), adds the field "length" to the object info, and provides convenient methods for string processing.

```
new string $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	String value (default blank).

The method *length* returns the string length. This is the same as "`$stringObj info length`".

```
$stringObj length
```

The method *append* simply appends values to the string.

```
$stringObj append $value ...
```

$value ...	String values to append.

The method "`@`" can be used for string indexing and modification. If indexing, it will return the value, and if modifying, it will return the object.

```
$stringObj @ $first <$last> <= $newstring>
```

$first	First index in range.
$last	Last index in range (default `$first`).
$newstring	Value to replace with. Default just returns the string index/range.

Example 10: String type example

Code:

```
new string x hello
$x append { world}
puts [$x length]
[$x @ 0 = H] print
```

Output:

```
11
Hello world
```

Type "bool" (boolean)

The type "*bool*" validates input by passing it through the Tcl *::tcl::mathfunc::bool* command, which ensures that the input is a valid boolean (0 or 1).

```
new bool $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	Boolean value (default 0).

The method "*?*" provides a shorthand if-statement control flow method.

```
$boolObj ?  $body1 <":" $body2>
```

$body1	Body to evaluate if boolean is true.
$body2	Body to evaluate if boolean is false (optional, required with ":" keyword).

Example 11: Boolean type example

Code:

```
# Procedure with type validation
proc foo {a b c} {
    new string a $a
    new string b $b
    new bool c $c
    $c ? $a : $b
}
puts [foo hello world true]; # hello
puts [foo hello world false]; # world
```

Output:

```
hello
world
```

Type "int" (integer)

The type "*int*" validates that the input is a valid integer, and additionally has increment/decrement methods.

```
new int $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	Integer value (default 0).

The methods "*++*" and "*--*", simply increment/decrement the integer object by 1, and return the object.

```
$intObj ++
```

```
$intObj --
```

The methods "*+=*", "*-=*", "**=*" and "*/=*" add, subtract, multiply, and perform integer division on the current value of the object variable. Like the *:=* method, it returns the object name.

```
$intObj += $expr
```

```
$intObj -= $expr
```

```
$intObj *= $expr
```

```
$intObj /= $expr
```

$expr	Expression to evaluate (passes through the "*:=*" method).

Example 12: Integer example

Code:

```
for {new int i} {[$i] < 4} {$i ++} {
    $i print
}
```

Output:

```
0
1
2
3
```

Type "float" (double-precision floating-point decimal)

The type "*float*" validates that input is a double-precision floating-point number, passing input through the Tcl *::tcl::mathfunc::double* command.

```
new float $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	Float value (default 0.0).

The methods "*+=*", "*-=*", "**=*" and "*/=*" add, subtract, multiply, and perform division on the current value of the object variable. Like the *:=* method, it returns the object name.

```
$floatObj += $expr
```

```
$floatObj -= $expr
```

```
$floatObj *= $expr
```

```
$floatObj /= $expr
```

$expr	Expression to evaluate (passes through "*:=*" method).

Example 13: Float example

Code:

```
# Harmonic mean of two numbers (converts to float)
proc hmean {x y} {
    new float x $x
    new float y $y
    [new float z] := {2*[$x]*[$y]}
    if {[$z] != 0} {
        $z /= {[$x] + [$y]}
    }
    return [$z]
}
puts [hmean 1 2]; # 1.3333
```

Output:

```
1.3333333333333333
```

Type "list"

The type "*list*" validates that the input is a list, adds the field "length" to the info dictionary, and additionally passes input from the operators ".=", ":=", and "::=" through the commands *lop*, *lexpr*, and *leval*. This variable type does not validate the values stored in the list, just that it is a valid list.

```
new list $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	List value (default blank).

The method *length* returns the list length. This is the same as "`$listObj info length`".

```
$listObj length
```

The method *append* simply appends values to the list.

```
$listObj append $value ...
```

$value ...	String values to append.

Example 14: List example

Code:

```
new list x foo
$x append bar
puts [$x info]
```

Output:

```
type list length 2 value {foo bar}
```

List Math Operations

The command *lop* performs simple math over lists, and is also built-in to the "`.=`" list operator.

```
lop $list $op <$arg ...>
```

`$list`	List value to map over.
`$op`	Tcl math operator.
`$arg ...`	Additional mathop arguments.

Example 15: Element-wise operations

Code:

```
# Primative variable method
set x {1 2 3}
set x [lop $x * 2]
puts $x
# Object variable method
new list x {1 2 3}
$x .= {* 2}
$x print
```

Output:

```
2 4 6
2 4 6
```

Example 16: Nested list operations

Code:

```
# Primative variable method
set x {1 2 3}
set x [lop [lop [lop $x + 1] * 2] - 3]
puts $x
# Object variable method
new list x {1 2 3}
[[$x .= {+ 1}] .= {* 2}] .= {- 3}
$x print
```

Output:

```
1 3 5
1 3 5
```

List Reference Mapping

Using the *::vutil::RefSub* syntax, you can create powerful wrapper commands. As a demonstration, this package provides two simple list object mapping commands: *leval* and *lexpr*, which are built-in to the "`:=`" and "`::=`" list operators.

```
leval $body <"-->" $refName>
```

```
lexpr $expr <"-->" $refName>
```

`$body`	Tcl script with list object references.
`$expr`	Tcl expression with list object references.
`$refName`	Optional reference variable to tie resulting list to. Blank to return value.

Example 17: Element-wise expressions

Code:

```
new list x {1 2 3}
new list y {4 5 6}
lexpr {$@x + $@y} --> z
$z := {double($@z)}
$z print
```

Output:

```
5.0 7.0 9.0
```

Example 18: Zip a list together

Code:

```
new list list1 {a b c d}
new list list2 {1 2 3 4}
leval {list $@list1 $@list2} --> zipped
$zipped ::= {string toupper $@.}
$zipped print
```

Output:

```
{A 1} {B 2} {C 3} {D 4}
```

List indexing

The methods "`@`" and "`@@`" either index or modify the list, depending on whether an assignment operator is used. If indexing, it calls *lindex* or *lrange*, and returns the corresponding value or range. If modifying with an assignment operator, it modifies the list with *lset* or *lreplace* and returns the object name. The "`=`" operator simply modifies the list directly. The other assignment operator create a temporary list variable for the specified index/range and evaluates the assignment operator on the temporary list. Then, it modifies the main list with the resulting value of the temporary list, and finally returns the object name.

```
$listObj @ $i ...  <$op $arg>
```

```
$listObj @@ $first $last <$op $arg>
```

`$i ...`	List indices.
`$first`	First index in range.
`$last`	Last index in range.
`$op $arg`	Assignment operator (`= .= := ::=`) and corresponding input.
`= $value`	Value to set.
`.= $oper`	Tcl math operator and additional mathop arguments, i.e. `"+ 5"`.
`:= $expr`	Tcl expression to evaluate.
`::= $body`	Tcl script to evaluate.

Example 19: List indexing

Code:

```
new list list1 ""
$list1 @ 0 = "hey"
$list1 @ 1 = "there"
$list1 @ end+1 = "world"
set a 5
$list1 @ end+1 := {$a * 2}
$list1 @@ 0 1 ::= {string totitle $@.}
$list1 print
```

Output:

```
Hey There world 10
```

Type "dict" (dictionary)

The type "*dict*" validates that the input is a Tcl dictionary, and provides methods for all the Tcl dictionary subcommands, with the exception of *info*, which due to a conflict with the standard method *info*, was replaced with *$dictObj stats*. Additionally, it adds the field "size" to the variable info. This package only provides a new, more convenient way to access the Tcl dictionary data structure, which is fully explained and documented here: https://www.tcl.tk/man/tcl/TclCmd/dict.html.

```
new dict $refName <$value>
```

$refName	Name of reference variable to tie to object.
$value	Dictionary value (default blank).

Below is an example of how the dictionary variable type can streamline basic dictionary queries.

Example 20: Dictionary example

Code:

```
# Create dictionary record
new dict record {
    name {John Doe}
    address {
        streetAddress {123 Main Street}
        city {New York}
    }
    phone {555-1234}
}
# Get values
puts [$record size]; # Number of keys (3)
puts [$record get name]; # John Doe
# Set/unset and get
$record set address street [$record get address streetAddress]
$record unset address streetAddress
puts [$record get address street]; # 123 Main Street
puts [$record exists address streetAddress]; # 0
```

Output:

```
3
John Doe
123 Main Street
0
```

Dictionary Access Methods

The method *get* is equivalent to the Tcl command *dict get*. It retrieves values from the dictionary at the specified key sequence, and returns an error if entry does not exist.

```
$dictObj get <$key ...>
```

`$key...`	Sequence of keys to query.

The method *exists* is equivalent to the Tcl command *dict exists*. It returns a boolean value indicating whether a value exists at the specified key sequence.

```
$dictObj exists $key <$key ...>
```

`$key...`	Sequence of keys to query.

The method *keys* is equivalent to the Tcl command *dict keys*. It retrieves a list of the keys stored in the dictionary, with an optional "glob" pattern.

```
$dictObj keys <$globPattern>
```

`$globPattern`	Optional pattern to match keys with. Default all keys.

The method *values* is equivalent to the Tcl command *dict values*. It retrieves a list of the values stored in the dictionary, with an optional "glob" pattern.

```
$dictObj values <$globPattern>
```

`$globPattern`	Optional pattern to match values with. Default all values.

The method *size* is equivalent to the Tcl command *dict size*. This is the same as "`$dictObj info size`". It simply returns the number of key-value pairings in the dictionary.

```
$dictObj size
```

The method *stats* is equivalent to the Tcl command *dict info*. This is the only method that is renamed from the Tcl subcommand counterpart, due to a conflict with the standard object variable method *info*. It simply returns data about the implementation of the dictionary datatype.

```
$dictObj stats
```

Dictionary Modification Methods

The method *set* is equivalent to the Tcl command *dict set*, except that it returns the object name. It assigns values to the dictionary at the specified key sequence.

```
$dictObj set $key <$key ...> $value
```

$key...	Sequence of keys to set.
$value	Value to set at key sequence.

The method *unset* is equivalent to the Tcl command *dict unset*, except that it returns the object name. It is the counterpart to *$dictObj set*, and unsets the dictionary entry at the specified key sequence.

```
$dictObj unset $key <$key ...>
```

$key...	Sequence of keys to unset.

The method *replace* is equivalent to the Tcl command *dict replace*, except that it modifies the object and returns the object name. It is similar to *$dictObj set*, except that multiple key-value pairings can be specified (or none).

```
$dictObj replace <$key $value...>
```

$key...	Key(s) to replace at.
$value...	Value(s) to replace with.

The method *remove* is equivalent to the Tcl command *dict remove*, except that it modifies the object and returns the object name. It is similar to *$dictObj unset*, except that multiple keys can be specified (or none).

```
$dictObj remove <$key ...>
```

$key...	Key(s) to remove

The method *merge* is equivalent to the Tcl command *dict merge*, except that it returns the object name. It merges the entries from multiple dictionaries into the dictionary object variable.

```
$dictObj merge <$dictionaryValue ...>
```

$dictionaryValue ...	Dictionary values to merge into object dictionary.

Dictionary Entry Manipulation Methods

The method *append* is equivalent to the Tcl command *dict append*, except that it returns the object name. It treats the dictionary entry value as a string, and appends to the string.

```
$dictObj append $key <$string ...>
```

$key	Key to append at.
$string ...	Text to append to dictionary entry.

The method *lappend* is equivalent to the Tcl command *dict lappend*, except that it returns the object name. It treats the dictionary entry value as a list, and appends to the list.

```
$dictObj lappend $key <$value ...>
```

$key	Key to lappend at.
$value ...	Values to append to list at specified key.

The method *incr* is equivalent to the Tcl command *dict incr*, except that it returns the object name. It treats the dictionary entry value as an integer, and increments the value.

```
$dictObj incr $key <$incr>
```

$key	Key to increment at.
$incr	Increment value. Default 1.

The method *update* is equivalent to the Tcl command *dict update*. It allows for modification of dictionary entries directly using a Tcl script.

```
$dictObj update $key $varName <$key $varName ...> $body
```

$key...	Keys to update.
$varName...	Variables to store values at specified keys.
$body	Script to evaluate.

Dictionary Mapping Methods

The method *filter* is equivalent to the Tcl command *dict filter*, except that it modifies the object and returns the object name. It removes entries that do not fit the specified criteria.

```
$dictObj filter $filterType $arg <$arg ...>
```

`$filterType`	Dictionary filter type.
`$arg ...`	Additional arguments. See documentation for Tcl command *dict filter*.

The method *for* is equivalent to the Tcl command *dict for*. It simply loops over the dictionary and evaluates a script.

```
$dictObj for "$keyVariable $valueVariable" $body
```

`$keyVariable`	Variable to store dictionary key in.
`$valueVariable`	Variable to store dictionary entry values in.
`$body`	Tcl script to evaluate for each dictionary entry.

The method *map* is equivalent to the Tcl command *dict map*, except that it modifies the object and returns the object name. It is equivalent to *$dictObj for*, except that the result of each iteration gets stored in the dictionary at the corresponding key.

```
$dictObj map "$keyVariable $valueVariable" $body
```

`$keyVariable`	Variable to store dictionary key in.
`$valueVariable`	Variable to store dictionary entry values in.
`$body`	Tcl script to evaluate for each dictionary entry.

The method *with* is equivalent to the Tcl command *dict with*. It sets variables with names equal to the dictionary key names, and evaluates a script. Note that this should not be used if the dictionary key names conflict with the object variable name.

```
$dictObj with <$key ...> $body
```

`$key...`	Sequence of keys containing the dictionary to use.
`$body`	Tcl script to evaluate with entries as variables.

Command Index