Tcl Variable Utilities

Version 0.11

Alex Baker

https://github.com/ambaker1/vutil

August 1, 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.

Initializing Local Namespace Variables

The command *local* is the counterpart to the Tcl *global* command, and creates local variables linked to variables in the current namespace, by simply calling the Tcl *variable* command multiple times.

```
local $name1 $name2 ...
```

$name1 $name2 ... Name(s) of variables to initialize

Example 1: Access namespace variables in a procedure

Code:

```
# Define global variables
global a b c
set a 1
set b 2
set c 3
namespace eval ::foo {
    # Define local variables
    local a b c
    set a 4
    set b 5
    set c 6
}
proc ::foo::bar1 {} {
    # Access global variables
    global a b c
    list $a $b $c
}
proc ::foo::bar2 {} {
    # Access local variables
    local a b c
    list $a $b $c
}
puts [::foo::bar1]; # global a b c
puts [::foo::bar2]; # local a b c
```

Output:

```
1 2 3
4 5 6
```

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 2: 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. This is especially useful for controlling a parameter study of an analysis from a top-level. If attempting to modify a locked variable, it will throw a warning, but not an error. You can lock array elements, but not an entire array.

```
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 3: 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
```

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 is separate from lock; a tie will override a lock, and a lock will override a tie. You can tie array elements, but not an entire array.

```
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 4: 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"
```

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 "**&**". This shared reference, regardless of scope, can be accessed with the command "***$&***".

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

$arg ...	Arguments for object.

Reference variables can also be referred to with the "**$@ref**" syntax in the context of iterator functions. The engine for this is the command *refsub*, which performs "**$@ref**" substitution on a given string, returning the updated string and all matched reference names. For example, "**$@ref**" is converted to "**${$@(ref)}**". To escape a reference, especially for nested substitution, simply add more "**@**" symbols, like "**$@@ref**".

```
refsub $string
```

Example 5: Reference Variable Substitution

Code:

```
lassign [refsub {$@& + $@x(1) $@@y}] string refs
puts $string
puts $refs
```

Output:

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

A simple example of this is provided for the *list* type, with the commands *leval* and *lexpr*.

Garbage Collection Superclass

The class *gcoo* 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. The constructor is configured as shown below, and ties the object to the specified reference variable, using *tie*.

```
::vutil::gcoo new $refName
::vutil::gcoo create $name $refName
```

$refName	Name of reference variable for garbage collection.
$name	Name of object.

In addition to tying the object to a reference variable in the superclass constructor, the *::vutil::gcoo* superclass also provides a method for copying the object to a new reference variable: "`-->`".

```
$gcooObj --> $refName
```

$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 6: Creating a class with garbage collection

Code:

```
oo::class create container {
    superclass ::vutil::gcoo
    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]
```

Output:

```
hello world
```

Variable-Object Links

The command *link* links a global variable to a TclOO object, using the name of the object as the variable name. The value of the linked object variable is accessed by calling the TclOO object with no arguments (the "unknown" method), and writing to the object-variable calls the object's "=" method. Unsetting the linked object variable also calls the object's "destroy" method, and destroying the object unsets the linked object variable. Linked object-variables are unlinked when the object is destroyed, but can also be unlinked with the command *unlink*.

```
link $object
```

```
unlink $object ...
```

$object ...	Object(s) to link/unlink.

Example 7: Linking an object variable

Code:

```
::oo::class create number {
    variable value
    constructor {args} {
        set value [uplevel 1 expr $args]
    }
    method unknown {args} {
        if {[llength $args] == 0} {
            return $value
        }
        next {*}$args
    }
    unexport unknown
    method = {args} {
        set value [uplevel 1 expr $args]
    }
    export =
}
link [tie a [number new 5]]; # garbage collection and obj-var link
puts [$a]; # 5
$a = 10 * [$a]
puts [$a]; # 50
incr $a
puts [subst $$a]; # 51
```

Output:

```
5
50
51
```

Object Variable Class

The TclOO class *var* is a subclass of *::vutil::gcoo*, that also sets up an object variable link with *link*. So, in addition to the copy method "`-->`", object variables can be manipulated directly with Tcl commands, and calling the object variable directly as a command with no arguments (e.g. `[$varObj]`) returns the object variable value.

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

`$objName`	Explicit name for object.
`$refName`	Name of reference variable.
`$value`	Value to set object variable to.

Example 8: Object variables with garbage collection

Code:

```
# Example showing how object variables behave in procedures
proc foo {value} {
    # Create named object with reference variable "result"
    var create myObj result $value
    append $result { world}
    return [list $result [$result]]; # Returns name and value of object
}
set result [foo hello]; # Not the same "result"
lassign $result name value
puts $name; # ::myObj
puts $value; # hello world
puts [info object isa object $name]; # 0 (object was deleted when procedure returned)
```

Output:

```
::myObj
hello world
0
```

Object Variable Methods

Additional information about the object variable can be accessed with object variable methods:

The method *info* accesses all object variable metadata. Fields "exists" and "type" always exist, and "value" is populated when the variable is initialized.

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

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

The method *print* is a short-hand way to print the value of the variable to screen.

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

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

Example 9: Printing the value of a variable

Code:

```
var new x {Hello World}
puts [$x info]
$x print
```

Output:

```
exists 1 type var value {Hello World}
Hello World
```

Object Variable Operators

In addition to being able to manipulate object variables directly with Tcl commands, variables can also be manipulated with object variable operators.

The operators "`=`" and "`:=`" assign the value of the object variable, and return the name of the object.

```
$varObj = $value
```

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

`$value`	Value to set object variable to.
`$expr`	Math expression to evaluate and set as object value.

The operator "`<-`" assigns the value of the object directly from another object variable of the same class, and, like the operator "`=`", returns the name of the object.

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

`$otherVarObj`	Other object variable to assign value from.

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

Example 10: Object variable manipulation features

Code:

```
var new x; # Create blank variable $x
[$x --> y] = 5; # Copy $x to $y, and set to 5
[var new z] <- [$x <- $y]; # Create $z and set to x after setting $x to $y.
$z := {[$z] + [$x]}; # Add $x to $z
append $y [set $x 0]; # Append $y the value of $x after setting $x to 0
puts [list [$x] [$y] [$z]]
```

Output:

```
0 50 10
```

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.

To demonstrate how easy it is to create a type class, below is the code used to create the *string* type.

Example 11: Creating a simple type

Code:

```
type new string {
    method info {args} {
        set (length) [my length]
        next {*}$args
    }
    method length {} {
        string length $(value)
    }
    method @ {i} {
        string index $(value) $i
    }
    export @
}
```

Note that the value 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 three properties: (value), (exists), 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.

Example 12: Type assertion

Code:

```
proc foo {bar} {
    type assert list $bar
    $bar @ end
}
```

Creating Type Variables

Then, using the types defined by *type*, the command *new* creates a new variable object of a specified type.

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

$type	Name of type.
$refName	Name of reference variable to tie to object.
$value	Value to set object variable to.

Now you can easily create variables in Tcl with a specified type!

Example 13: Creating a new *string* object variable

Code:

```
new string x
set $x {hello world}
puts [$x length]
puts [$x info]
puts [$x @ end]
$x print
```

Output:

```
11
exists 1 length 11 type string value {hello world}
d
hello world
```

Type Library

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

Type "var"

The type "*var*" is just an alternative way to create an object variable (same syntax as "new" method for *var* class). It does not have any additional metadata or methods.

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

$refName	Name of reference variable to tie to object.
$value	Value to set object variable to.

Type "string"

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

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

$refName	Name of reference variable to tie to object.
$value	String value.

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

```
$stringObj length
```

The method "`@`" returns the character at the specified index.

```
$stringObj @ $i
```

$i	String index.

Type "bool" (boolean)

The type "*bool*" validates that the input is a valid boolean.

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

$refName	Name of reference variable to tie to object.
$value	Boolean value.

In addition to the standard object variable methods, the "bool" type 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 14: String and boolean example

Code:

```
# proc with types
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.

In addition to the standard assignment operators "`=`" and "`:=`", the "int" type provides the increment/decrement assignment operators "`+=`" and "`-=`".

```
$intObj $op $expr
```

`$op`	Assignment operator:
`+=`	Increments the variable by the value of `$expr`.
`-=`	Decrements the variable by the value of `$expr`.
`$expr`	Tcl math expression to evaluate.

There are also two short-hand increment/decrement operators, "`++`" and "`--`", which simply increment or decrement the integer object by 1.

```
$intObj ++
```

```
$intObj --
```

Example 15: Integer example (for loop)

Code:

```
for {new int i 0} {[$i] < 3} {$i ++} {
    puts [$i]
}
```

Output:

```
0
1
2
```

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.

In addition to the standard assignment operators "=" and ":=", the "float" type provides the following:

```
$floatObj $op $expr
```

$op	Assignment operator:
+=	Adds the value of **$expr** to the variable.
-=	Subtracts the value of **$expr** from the variable.
***=**	Multiplies the variable by the value of **$expr**.
/=	Divides the variable by the value of **$expr**.
$expr	Tcl math expression to evaluate.

Example 16: Float example (procedure with type assertion)

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, and additionally passes input from its main "`:=`" operator through the *lexpr* command.

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

`$refName`	Name of reference variable to tie to object.
`$value`	List value.

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

```
$listObj length
```

The method "`@`" acts as either list indexing or list setting, depending on whether the "`=`" or "`:=`" keywords are used. If indexing, it returns the value at the specified index. If setting, it sets the value and returns the object name.

```
$listObj @ $i ...  <"=" $value | ":=" $expr>
```

`$i ...`	List indices.
`$value`	Value to set.
`$expr`	Math expression, passed through *lexpr* command.

Example 17: List example

Code:

```
[new list list1] = {hello world}
puts [$list1 length]; # 2
$list1 @ 0 = "hey"
$list1 @ 1 = "there"
$list1 @ end+1 = "world"
puts [$list1 @ end]; # world
set a 5
$list1 @ end+1 := {$a + 1}
puts [$list1 info]; # exists 1 length 4 type list value {hey there world 6}
```

Output:

```
2
world
exists 1 length 4 type list value {hey there world 6}
```

Type "dict"

The type "*dict*" validates that the input is a Tcl dictionary, and provides methods for getting/setting/unsetting dictionary values, checking if values exist, and getting the dictionary size. Additionally, it adds the field "size" to the variable info.

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

$refName	Name of reference variable to tie to object.
$value	Dictionary value.

The method $dictObj size returns the dictionary size. This is the same as "**$dictObj info size**".

```
$dictObj size
```

The methods *set* and *unset* set and unset values in the dictionary, and return the object name. The method *get* returns values in the dictionary, and the method *exists* returns whether the key pairing exists.

```
$dictObj set $key ...  $value
```

```
$dictObj unset $key ...
```

```
$dictObj get $key ...
```

```
$dictObj exists $key ...
```

$key ...	Dictionary keys.
$value	Value to set.

Below is an example of how the dictionary variable type can streamline basic dictionary queries. Note that not all of the subcommands of the Tcl *dict* are provided as dictionary object methods, but because it is an object variable, it can still be manipulated using normal *dict* commands as well.

Example 18: Dictionary example

Code:

```
# Create dictionary record
new dict record {
    name {John Doe}
    address {
        streetAddress {123 Main Street}
        city {New York}
        state {NY}
        zip {10001}
    }
    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
# Manipulate with normal dict commands
dict lappend $record name Smith
puts [$record get name]
```

Output:

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

List Object Mapping

Using the *refsub* syntax, you can create powerful wrapper commands. As a demonstration, this package provides two simple list object mapping commands: *leval* and *lexpr*. The *lexpr* command additionally is built-in to the "`:=`" operator for the *list* type.

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

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

`$body`	Body with list object references.
`$expr`	Tcl math expression with list object references.
`$refName`	Optional reference variable to tie resulting list to.

Example 19: Element-wise math

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 20: Zip a list together (modified from https://www.tcl-lang.org/man/tcl/TclCmd/lmap.htm)

Code:

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

Output:

```
{a 1} {b 2} {c 3} {d 4}
```

Command Index