Tcl Variable Utilities

Version 2.0

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. 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 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
```

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. 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 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"
```

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 “`${::@(ref)}`”. To escape a reference, especially for nested substitution, simply add more “`@`” symbols, like “`$@@ref`”. This command is not exported, and 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>
```

`$refName`	Name of reference variable.
`$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.
`$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.

Check out the example below for some examples of how you can use these features to modify 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* is a alternative 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.

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 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.

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: Creating a new *string* object variable

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 that the input is a valid boolean, passing input through the Tcl *::tcl::mathfunc::bool* command.

```
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
```

Command Index