Tcl Variable Utilities

Version 4.0.1

Alex Baker

https://github.com/ambaker1/vutil

May 20, 2024

Abstract

The “vutil” package provides advanced functionality for working with variables in Tcl, such as read-only variables and a comprehensive TclOO garbage collection framework.

This package is also a Tin package, and can be loaded in as shown below:

Example 1: Installing and loading “vutil”

Code:

```
package require tin
tin add -auto vutil https://github.com/ambaker1/vutil install.tcl 4.0-
tin import vutil
```

Default Variable Values

The command *default* assigns a default value to a variable if it does not exist. This can be used to write scripts that behave like procedures with default values.

```
default $varName $value
```

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

The examples below shows how default values are only applied if the variables do not exist.

Example 2: Variable defaults

Code:

```
set a 5
default a 7
puts $a
unset a; # variable no longer exists
default a 7
puts $a
```

Output:

```
5
7
```

Example 3: Overriding default values in file 'putsMessage.tcl'

Code:

```
source putsMessage.tcl
set message {hello world}
source putsMessage.tcl
```

Output:

```
foo bar
hello world
```

Example 4: File 'putsMessage.tcl'

Code:

```
default message {foo bar}
puts $message
```

Read-Only Variables

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. This can be used to override values in a script from a top-level, allowing the user to run a parameter study without modifying the source scripts.

```
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 5: 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 $varName <$object>
```

`$varName`	Name of variable for garbage collection.
`$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*. Renaming or destroying an object also unties all variables tied to it.

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

`$name1 $name2 ...`	Variables to untie.

Example 6: Variable-object ties

Code:

```
oo::class create foo {
    method sayhello {} {
        puts {hello world}
    }
}
tie a [foo create bar]
set b $a; # object alias
$a sayhello
$b sayhello
unset a; # destroys object
$b sayhello; # throws error
```

Output:

```
hello world
hello world
invalid command name "::bar"
```

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

Garbage Collection Superclass

The class *::vutil::GarbageCollector* is a TclOO superclass that includes garbage collection by tying the object to a specified variable using *tie*. This class is not exported.

Below is the syntax for the superclass constructor.

```
::vutil::GarbageCollector new $varName
```

```
::vutil::GarbageCollector create $name $varName
```

`$varName`	Name of variable for garbage collection.
`$name`	Name of object (for “create” method).

In addition to tying the object to a variable in the constructor, the *::vutil::GarbageCollector* superclass provides a public copy method: “`-->`”, which calls the private method *CopyObject*.

```
$gcObj --> $varName
```

```
my CopyObject $varName
```

`$varName`	Name of variable for garbage collection.

Below is an example of how this superclass can be used to build garbage collection into a TclOO class. This process is formalized with the superclass *::vutil::ValueContainer*.

Example 7: Simple value container class

Code:

```
oo::class create value {
    superclass ::vutil::GarbageCollector
    variable myValue
    method set {value} {set myValue $value}
    method value {} {return $myValue}
}
[value new x] --> y; # create x, and copy to y.
$y set {hello world}; # modify $y
unset x; # destroys $x
puts [$y value]
```

Output:

```
hello world
```

Container Superclass

The class *::vutil::ValueContainer* is a TclOO superclass, built on-top of the *::vutil::GarbageCollector* superclass. In addition to the copy method "`-->`", this class stores a value in the variable "myValue", which can be accessed with the methods *GetValue* and *SetValue*. This class is not exported.

Below is the syntax for the superclass constructor.

```
::vutil::ValueContainer new $varName <$value>
```

```
::vutil::ValueContainer create $name $varName <$value>
```

`$name`	Name of object (for "create" method).
`$varName`	Name of variable for garbage collection.
`$value`	Value to store in object. Default blank.

Getting and Setting

Calling the object by itself calls the *GetValue* method, which simply queries the value in the container. The assignment operator, "`=`", calls the *SetValue* method, which sets the value in the container.

```
$vcObj = $value
```

```
my SetValue $value
```

`$value`	Value to store in container.

Example 8: Simple container

Code:

```
::vutil::ValueContainer new x
$x = {hello world}
puts [$x]
```

Output:

```
hello world
```

Math Assignment Operator

The math assignment operator, "`:=`", calls the *SetValue* method after evaluating the expression passed through the Tcl *expr* command.

```
$vcObj := $expr
```

`$expr`	Expression to evaluate and assign to object.

The math assignment operator makes use of the private method *Uplevel*, which evaluates the body of script at a specified level, while making the object command name available through the alias "`$.`". This can be nested, as it restores the old alias after evaluation.

```
my Uplevel $level $body
```

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

`$arg ...`	Method arguments for object.

Example 9: Modifying a container object

Code:

```
[::vutil::ValueContainer new x] = 5.0
$x := {[$.] + 5}
puts [$x]
```

Output:

```
10.0
```

Advanced Operators

The pipe operator, "`|`", calls the *TempObject* method, which copies the object and evaluates the method, returning the result or the value of the temporary object if the result is the object.

```
$vcObj | $method $arg ...
```

```
my TempObject $method $arg ...
```

`$method`	Method to evaluate in temporary object.
`$arg ...`	Arguments for method.

The ampersand operator "`&`", calls the *RefEval* method, which copies the value to a variable, and evaluates a body of script. The changes made to the variable will be applied to the object, and if the variable is unset, the object will be deleted. Returns the result of the script.

```
$vcObj & $varName $body
```

```
my RefEval $varName $body
```

`$varName`	Variable name to use for reference.
`$body`	Body to evaluate.

Example 10: Advanced methods

Code:

```
[::vutil::ValueContainer new x] = {1 2 3}
# Use ampersand method to use commands that take variable name as input
$x & ref {
    lappend ref 4
}
puts [$x | = {hello world}]; # operates on temp object
puts [$x]
```

Output:

```
hello world
1 2 3 4
```

Example 11: Using the value container superclass to create a vector class

Code:

```
# Create a class for manipulating lists of floating point values
oo::class create vector {
    superclass ::vutil::ValueContainer
    variable myValue; # Access "myValue" from superclass
    method SetValue {value} {
        # Convert to double
        next [lmap x $value {::tcl::mathfunc::double $x}]
    }
    method print {args} {
        puts {*}$args $myValue
    }
    method += {value} {
        set myValue [lmap x $myValue {expr {$x + $value}}]
        return [self]
    }
    method -= {value} {
        set myValue [lmap x $myValue {expr {$x - $value}}]
        return [self]
    }
    method *= {value} {
        set myValue [lmap x $myValue {expr {$x * $value}}]
        return [self]
    }
    method /= {value} {
        set myValue [lmap x $myValue {expr {$x / $value}}]
        return [self]
    }
    method @ {index args} {
        if {[llength $args] == 0} {
            return [lindex $myValue $index]
        } elseif {[llength $args] != 2 || [lindex $args 0] ne "="} {
            return -code error "wrong # args: should be\
                    \"[self] @ index ?= value?\""
        }
        lset myValue $index [::tcl::mathfunc::double [lindex $args 1]]
        return [self]
    }
    export += -= *= /= @
}
# Create a vector
vector new x {1 2 3}
puts [$x | += 5]; # perform operation on temp object
[$x += 5] print; # same operation, on main object
puts [$x @ end]; # index into object
```

Output:

```
6.0 7.0 8.0
6.0 7.0 8.0
8.0
```

Command Index