Tcl Variable Utilities

Version 3.1

Alex Baker

https://github.com/ambaker1/vutil

October 22, 2023

Abstract

The "vutil" package provides utilities such as read-only variables and TclOO garbage collection.

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 3.0-
tin import vutil
```

Default Variable Values

The command *default* assigns a default value to a variable if it does 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
```

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.

```
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
```

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*.

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

$name1 $name2 ...	Variables to untie.

Example 4: 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::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 variable using *tie*. Below is the syntax for the superclass constructor.

```
::vutil::GC new $varName
```

```
::vutil::GC 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::GC* superclass also provides a public copy method that sets up garbage collection: "-->", 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.

Example 5: Simple container class

Code:

```
oo::class create value {
    superclass ::vutil::GC
    variable myValue
    constructor {varName {value {}}} {
        set myValue $value
        next $varName
    }
    method set {value} {set myValue $value}
    method value {} {return $myValue}
}
value new x {hello world}; # create new value, tie to x
[$x --> y] set {foo bar}; # copy to y, set y to {foo bar}
puts [$x value]
puts [$y value]
```

Output:

```
hello world
foo bar
```

Container Superclass

The class *::vutil::Container* is a TclOO superclass, built on-top of the *::vutil::GC* superclass. In addition to the copy method "-->", this class stores a value in the variable "self", which can be accessed with the methods *GetValue* and *SetValue*. This class is not exported, and not intended for direct use, but rather is a template for container classes. Below is the syntax for the superclass constructor.

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

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

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

Calling a container object by itself calls the *GetValue* method, which queries the value in the container.

```
$containerObj
```

```
my GetValue
```

The assignment operator, "=", calls the *SetValue* method, which sets the value in the container.

```
$containerObj = $value
```

```
my SetValue $value
```

$value	Value to store in container.

The pipe operator, "/", calls the *TempObject* method, which copies the object to a temporary object, evaluates the method, and returns the result, or the temporary object value if the result is the temporary object.

```
$containerObj | $method $arg ...
```

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

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

Example 6: Advanced container class

Code:

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