Using Tcl 8.5 Features to Build an OO System

Donal K. Fellows <donal.k.fellows@manchester.ac.uk>

In this paper I aim to show that object-oriented programming using Tcl is better supported in the upcoming 8.5 release than at any time before. To show this, I identify the key features of a basic OO system and illustrate mechanisms in Tcl 8.5 that support them.

Introduction

What is an object-oriented programming system? Crudely put, at the basic level it is just a scheme for joining data and the operations that work on that data together in a single parcel. For something to be an object, it therefore requires a few things:

- Data parcelled together,
- A reference to the parcel,
- A scheme for going from the parcel-reference to the operations on that data, and
- A way of making sure that when the parcel-reference goes away (whether this happens through explicit or implicit deletion) so does the data parcel itself.

This is all very well, but what does this mean when applied to Tcl? What features of the language are particularly suited to each of these major requirements?

I also discuss how to take this core OO system and build it up into something more similar to more common OO schemes.

An Object Engine Core

Basic Data Model

For parcelling data together it is nice to use Tcl 8.5's new dictionary value type. A dictionary is a *container value*, just like a list, except that its elements are addressable by name and not position. In many ways (especially when placed in a variable), it is like an array, but it allows for things like arbitrary nesting, trivial writing of the whole collection of values to a channel, and simple transfer of the values between procedures. It is also considerably faster.

Indeed, it is best to use a dictionary to hold the state of whole groups of objects. By storing all object state in an array in either the class or the OO package's own namespace, the natural nest-ability of dictionaries can be leveraged to allow us to avoid having to work with composite array element names or other such tricks.

To keep things simple in this paper, I shall store object states as dictionaries held in a global array.

```
set states($objName) [dict create \
      foo bar \
      language Tcl \
      bandersnatch {frumious shunned}]
```

Later, when I discuss methods, I shall show how to use the facilities of 8.5's dictionaries to make referring to the object state very easy indeed.

Object References

The natural object reference in Tcl is the handle (not the Tcl_Obj, which is really a basic value and not an object), and the most common kind of handle is a command. This is what you see in many packages (for example, [incr Tcl], XOTcl, Snit, stooop, etc.) and also with several commands in the Tcl/Tk core (**interp** and all Tk widgets). The other kind of object handle is based on pure values, and this forms the basis of how objects such as I/O channels and HTTP tokens are managed.

I shall be using the command-like form here; after all, it is what most people seem to think of as being objects as represented in Tcl. The command name also makes for a suitable per-object token for use when looking up the state of the object. The only difficulty with this is dealing with command renaming, but this is also resolvable easily. All that is required is a call-back that is triggered whenever the object command is renamed; this is provided by setting a rename **trace**.

```
trace add command $objName rename \
      RenameObject
proc RenameObject {from to op} {
   global states
   set states($to) $states($from)
   unset states($from)
}
```

However, as it happens it is slightly easier to give each object a unique sequence number and use that number as the index into the *states* array. For convenience (and looking like other OO systems), I instead set the *this* key in the dictionary to be the name of the current command for the object, so rename traces are still useful.

```
dict set states($num) this $objName
trace add command $objName rename \
      [list RenameObject $num]
```

```
proc RenameObject {num from to op} {
   global states
   dict set states($num) this $to
}
```

Object Operations

Now we have our data and our command, the best way of joining the two is to use some of the advanced features of Tcl 8.5's ensemble commands.

Two of the advanced features of the **namespace ensemble** command are the **-command** and **-map** options. By specifying the command name ourselves, we can implement the handle name we choose directly as a command, and by specifying a subcommand-to-implementation mapping we have access to a powerful and fast rewrite engine that allow us to convert commands like:

```
objname subcmd arg1 arg2 arg3
```

into this:

```
impl objnumber arg1 arg2 arg3
```

by just including a single entry in the mapping that states that '*subcmd*' is implemented by '*impl objnumber*'. It is easy to derive this information from the method declarations in the class. As an additional bonus, the ensemble engine handles unique command prefixes for you.

```
namespace ensemble create \
    -command $objName -map [dict create\
        subcmd1 "impl1 $objnumber" \
        subcmd2 "impl2 $objnumber"]
```

Now that we know how the outline of we are going to handle methods, we need to look at how the methods will in turn access the state of the object. To do this, we use another feature of 8.5 (still in draft at the time of writing), the **dict with** subcommand. This "binds" variables to the keys of a dictionary for the duration of the execution of a user-supplied script, and writes any changes back to the dictionary when the script finishes. (There is an advanced version, **dict update**, that lets the user control what keys are bound and to what variables.)

The strategy for using **dict with** is to wrap the supplied script body for the method which is accessing the state to open out all the dictionary keys into local variables, so it ends up something like this:

```
dict with ::state($objnumber) {
   # The script body goes here, e.g.
   puts "this object is $this"
}
```

I provide four built-in methods as well, being *get*, which allows anyone to read a value from the object's state dictionary, *set*, which lets a value be inserted into or updated within the object's state dictionary, *unset*, which removes a key (and its value) from the dictionary, and *methods*, which gives access to the machinery for management of the object's methods (creating new ones somewhat like **proc**, describing whether a particular method is defined for the object, and providing a list of what methods are defined for the object.)

Object Deletion

The final component of our look at how modern Tcl supports object schemes is how to delete objects neatly. Since our objects are commands, it is natural to use the deletion of the command as our signal to dispose of the object. Luckily, we can just use command traces to detect when this event occurs and dispose of the object's state for us by calling **unset** on the array member variable holding the state.

```
trace add command $objName delete \
      [list DeleteObject $num]
proc DeleteObject {num args} {
   global state
   unset state($num)
}
```

Wrapping Up

Finally, there is just one more thing to do, and that is to decide on how to create our objects. To this end, I simply have a command that spits out new a new object every time it is asked. It handles the setting up of the traces, initialization of the state dictionary, and setting up the default methods so that the object can be customized for use.

```
proc NewObject {} {
   global state counter definedMethods
   ### Choose an object name
   set ob ::ob[incr counter]
   ### Set up the state
   set state($counter) [dict create \
         this $ob \
         {state reference} $counter]
   set definedMethods($counter) {}
   ### Initialise the methods
   set map {}
   foreach method {
      get set unset methods
   } {
      dict set map $method [list \
            ${method}Impl $counter]
   }
   ### Make the command
   namespace ensemble create \
           -command $ob -map $map
   ### Set up the traces
   trace add command $ob delete [list \
         DeleteObject $counter]
   trace add command $ob rename [list \
         RenameObject $counter]

   return $ob
}
```

The *definedMethods* array is used to keep track of what methods have been created using the *methods* method so that they can be correctly deleted when the object itself goes away. This can be seen in the way that the *methods* method is implemented.

(Also note the way in which the arguments are parsed; in particular the use of **lassign**, long a staple of TclX, to break up the argument list into formal parameters after parsing.)

```
proc methodsImpl {this args} {
   global state definedMethods
   set ob [dict get $state($this) this]
   set methods [namespace ensemble \
         configure $ob -map]
   ### General argument parsing
   if {[llength $args] == 0} {
      # Get list of all methods
      return [dict keys $methods]
   } elseif {[llength $args] == 1} {
      # Test for specific method
      return [dict exists $methods \
            [lindex $args 0]]
   } elseif {[llength $args] != 3} {
      # Boom!
      return -code error \
            "wrong # args: must be \
            \"$ob methods ?name? \
            ?arguments body?\""
   }
   ### Construct the implementation proc
   lassign $args name args body
   # Yes, that is a variable name with a
   # space in it.
   set bodypfx
      ";dict with ::oo::state($this) "
   uplevel \#0 [list proc $name \
      [linsert $args 0 {{this object}}]\
      $bodypfx$body]
   ### Splice into ensemble subcmd map
   namespace ensemble configure $ob \
         -map [dict replace $methods \
         $name [list $name $this]]
   lappend definedMethods($this) $name
}
```

Making More of Objects

Now we have a bare-bones object system, what can we do with it?

One thing that many people expect from their OO systems is classes. Their purpose is to issue many objects of essentially the same kind, all with the same methods. They also support things like inheritance, constructors, destructors, reference handling, etc.

A *Class* Object

The principle property of any class is its definition, as that encapsulates what it has as its superclasses, what its properties are, and what its methods are. We also need to have some way of describing these attributes and applying them when we create a new instance of the class. The easiest way of doing this is to represent the class definition as a Tcl script, of course, but that means we need to have an object-specific mechanism for mapping commands like *inherit*, *constructor*, etc. onto the behaviour we would like to use for them.

The easiest way of doing this is to use a temporary namespace. By creating suitable aliases in the namespace, we specialize the commands so that they insert methods and state dictionary entries correctly into the object being created. This even works for inheritance, because we just need to recursively process the definition from the superclass (while keeping a note of the superclass so we know the overall object's type hierarchy). So I set up the *method*, *constructor* and *destructor* aliases like this (excerpted from the *new* method of the *Class* object's instances):

```
interp alias {} ::ooinit::method \
      {} $theObject method
interp alias {} ::ooinit::constructor \
      {} ::ooinit::method constructor
interp alias {} ::ooinit::constructor \
      {} ::ooinit::method destructor {}
```

And I set up the *inherit* alias like this:

```
interp alias {} ::ooinit::inherit \
      {} ::SetupInherit $theObject
proc ::SetupInherit {obj superclass} {
   ### Add to classes if not there
   set classes [$obj get classes]
   if {!($superclass in $classes)} {
      lappend classes $superclass
      $obj set classes $classes
   }
   ### Install the superclass's defn
   uplevel 1 \
         [$superclass get definition]
}
```

With these aliases, all I need to do to set up the instance is execute its definition script in the correct namespace and then check whether a constructor has been defined:

```
Class method new {args} {
   ### Make the object
   set obj [NewObject]
   ### Seed the class hierarchy
   $obj set class $this
   $obj set classes [list $this]

   ### Set up the aliases here…

   ### Install the definition
   namespace eval ::ooinit \
         [$this get definition]
   $obj method delete {} {
      if {[$this method destructor]} {
         $this destructor
      }
      rename $this {}
   }
   ### Run the constructor, if defined
   if {[$obj method constructor]} {
      $obj constructor {expand}$args
   }
   return $obj
}
```

I handle the removal of objects by defining a *delete* method which calls the destructor (if it is defined) and then deletes the command using **rename**. This is installed last so classes can't (easily) prevent their instances from being deleted.

An *Object* Object

Now that we have a class, we should also define a root of the class hierarchy which has all the things in which we would like every class to have. I'll call this root class *Object*. In it, I provide six methods:

eq: An equality-test method for objects, defined through examining the basic counter reference embedded within the object. I use the counter reference because it is stable even when the command is renamed or referred to through **namespace export/import**.

isa: A method to test whether the object is of the given class.

<: A method that returns the counter reference embedded within the object. These are exposed so that collection classes can keep references that are valid across object renaming.

>: A method that returns the object with the given counter reference, since collections should not actually return object references. This method could be theoretically implemented as a utility procedure; it is just nice to keep everything in one place.

addRef: As with the Tcl_Obj structure in the Tcl core, the Object class's instances maintain a reference counter with about the same semantics. This method increments that counter, which starts at zero.

delRef: This method is the counterpart of the *addRef* method, and reduces the reference count of the object by one, calling the *delete* method when the reference count goes below 1.

To illustrate how these methods are defined, here is the full definition of the *isa* method:

```
method isa someClass {
   expr {$someClass in $classes}
}
```

This shows not only how succinct a method declaration can be, but it also illustrates the (proposed) new **in** operator in Tcl 8.5. Without that, the method would need to be written like this:

```
method isa someClass {
   expr {
      [lsearch -exact \
            $classes $someClass] >= 0
   }
}
```

As you can see, the **in** operator makes the code much clearer!

The last part of the *Object/Class* complex is splicing the *Class* class so that it looks like it is a subclass of *Object*. This has to be done especially because otherwise it is impossible to bootstrap *Class* (they circularly depend on each other.)

Demonstrating Object*: A List Collection Class*

To illustrate how the features of the *Object* scheme work together, here is the full text of a *List* class that acts as a collection of objects.

```
Class define List {
   ### List is a subclass of Object
   inherit Object
   ### Make a spot in the dict to hold
   ### our content list
   property content {}
   ### Constructor hands off to add
   constructor {args} {
      $this add {expand}$args
      # OK, [dict with] isn't perfect
      set content [$this get content]
   }
   ### A bunch of useful methods that
   ### implement the "List API"
   method add {args} {
      foreach obj $args {
         lappend content [$obj <]
         $obj addRef
      }
   }
   method length {} {
      llength $content
   }
   method index {idx} {
      $this > [lindex $content $idx]
   }
   method find {obj} {
      lsearch -exact $content [$obj <]
   }
   method remove {idx} {
      [$this > [lindex $content $idx]] \
            delRef
      # Tricky reference management!
      $this set content {}
      set content [lreplace \
            $content[set content {}] \
            $idx $idx]
   }
   # This method works by building a
   # command executed in the caller...
   method iterate {var body} {
      set body "set [list $var] \[\
            [list $this] > \
            $[list $var]]$body"
      uplevel 1 [list \
            foreach $var $content $body]
   }
   ### Release our refs on deletion
   destructor {
      foreach ref $content {
         [$this > $ref] delRef
      }
   }
}
```

Of particular note is the *remove* method, which uses the fact that concatenating a value with the empty string is a no-work operation in Tcl 8.5, allowing us to read the value out of the variable in such a way that we hold the only reference to its

Tcl_Obj and can use (behind the scenes) the efficient in-place version of **lreplace**.

If we want to produce a variation of the *List* that does not permit duplicates, we can do that quite simply like this:

```
Class define UniqueList {
   inherit List
   method add {args} {
      foreach obj $args {
         if {[$this find $obj] == -1} {
            lappend content [$obj <]
            $obj addRef
         }
      }
   }
}
```

To see how constructors and destructors can work together, here is an example class that uses them to print a message when an object is created and then repeat that message when the object is deleted;

```
Class define Example {
   inherit Object
   constructor {string} {
      puts "Made $this - $string"
      $this set initMsg $string
   }
   destructor {
      puts "Killing $this - $initMsg"
   }
}
```

Showing Off

Finally, let's show the output of a short session using these classes. Here's the script being executed:

```
package require oo

set o1 [Example new foo]
set o2 [Example new bar]
$o1 delete
$o2 delete

set l [List new]
$l add [Example new "Demo #1"]
$l add [Example new "Demo #2"]
$l add [Example new "Demo #3"]
rename [$l index 0] ::evil
$l add [$l index 0]
puts "Have [$l length] items in list"
set ctr 0
$l iterate v {
   puts -nonewline "Item [incr ctr]: $v"
   if {[$v isa Object]} {
      puts " (an object)"
   } else {
      puts " (not an object)"
   }
}
$l delete
```

This produces the output like the following (though particular object names are not guaranteed):

```
Made ::ob1 - foo
Made ::ob2 - bar
Killing ::ob1 - foo
Killing ::ob2 - bar
Made ::ob4 - Demo #1
Made ::ob5 - Demo #2
Made ::ob6 - Demo #3
Have 4 items in list
Item 1: ::evil (an object)
Item 2: ::ob5 (an object)
Item 3: ::ob6 (an object)
Item 4: ::evil (an object)
Killing ::ob5 - Demo #2
Killing ::ob6 - Demo #3
Killing ::evil - Demo #1
```

As you can see, the renaming of *::ob4* to *::evil* did not disrupt the fact that the *List* instance (which incidentally is called *::ob3* of course) contained it.

If we had used a *UniqueList* instead of a *List*, the second half of the output would have been this:

```
Have 3 items in list
Item 1: ::evil (an object)
Item 2: ::ob5 (an object)
Item 3: ::ob6 (an object)
Killing ::evil - Demo #1
Killing ::ob5 - Demo #2
Killing ::ob6 - Demo #3
```

Summary

As I have described, several of the features of Tcl 8.5 make doing a simple OO system really tremendously easy. The particular features that have been illustrated here are (with their TIP numbers):

- Dictionaries (TIP#111, draft TIP#212)
- Ensembles (TIP#112)
- Expanding substitutions (TIP#157)
- List membership operator (draft TIP #201)
- **lassign** command (TIP #57)

Note that some of these features are still in draft form at the time of writing. There is no guarantee that they will be in the final release of 8.5, though I would hope that will actually happen. For details of particular Tcl Improvement Proposals, see http://tip.tcl.tk/.

The other "new" feature illustrated here is the fact that concatenation of an empty string does not force the creation of the string representation of a Tcl_Obj value. This means that it is possible to construct (as shown) a very fast scheme for extracting an object from a variable and removing the reference from that variable at the same time. This represents a significant gain over the previous "idiom" in use, the **K** trick (and its pure-bytecode variants using **list** and **lindex**), because it is possible to do away with both memory allocation and the initialization of a procedure call-frame; it is just a peep-hole optimization in the implementation of INST_CONCAT.