Objects in Tcl

Abstract:

For a long time, Tcl did not have a native object model. But the flexibility of the language allows you to easily implement your own. We will also look briefly at some of the existing object-oriented extensions for Tcl.

Introduction

C++, C#, Python, Java and many other programming languages offer support for object oriented programming. Even without explicit language support, objects can easily be simulated. Object orientation is really just a way of thinking; it has more to do with design than with implementation. Look at the source code of the Tcl interpreter itself for a great example of object oriented programming in C (not C++!). If you are disciplined enough to always pass a pointer to a struct as the first parameter to a function, you can see such a function as a 'method' of the struct. You do not need actual language support to create object-oriented code. Even inheritance and polymorphism can be mimicked by clever use of function pointers.

Before version 8.6, Tcl did not offer object oriented primitives. This article presents a number of simple techniques to add objects to Tcl. Once you understand these techniques, you will be able to figure out most of the existing object packages and extensions.

Existing extensions

Before version 8.6, Tcl did not have a built-in object system. Over the years, people have filled this gap with a large number of extensions. Some of these extensions are written in C, and must be compiled to make them available in the Tcl language. A good example is [Incr Tcl], an extension that introduces primitives such as `class`, `method` and `constructor`.

Other extensions, such as Jean-Luc Fontaine's Stooop, are written in Tcl itself. They don't require any compilation. You may wonder how it is possible to extend Tcl with new primitives written in Tcl itself. This article answers that question by zooming in on the techniques of *object commands* and *class commands*. We will not discuss extensions that require compilation.

We should also distinguish between static and dynamic classes. With *static classes*, the members of a class cannot be changed at runtime. You *can* introduce new classes into a running system, but once a class is created, you cannot add new methods or data members to it. Similarly, you can create new instances of a class, but you cannot (easily) change the class of an existing instance, or give it instance-specific methods or member variables.

[Incr Tcl] is an example of a Tcl extension with static classes. You cannot add methods or variables to an existing class or object. You can, however, change the implementation of any method of a class (just like rewriting the body of an existing procedure in Tcl). And of course, you can inherit from an existing class and add new methods and variables in the derived class.

But since Tcl is a dynamic language, in which you can introduce new procedures and new variables at run-time, it seems more appropriate to also allow the creation of new methods and member variables at run-time. That requires a *dynamic class mechanism* such as offered by OTcl.

Both static and dynamic object-oriented extensions of Tcl can make use of the techniques described in this article.

A simple example

We will slowly introduce the necessary techniques, starting from an extremely simple example.

Suppose we want to write a script that stores information about fruit. We want to manipulate objects such as apples and lemons. We intend to give each object a unique identifier or number to distinguish it from other objects. We also want to store some properties for each object, such as their color.

As a first approach, we could store the object colors in a Tcl array, indexed by the object name. For example:

```
1 set a_color(a1) green
2 set a_color(a2) yellow
3 set a_color(a3) red
```

We now have three objects a1, a2, a3, each with their own color. Even an extremely simple approach like this one is already useful in many cases where you need to map object attributes to their values.

We can make this more attractive by hiding the array, using two access procedures:

example_01/apples.tcl

```
 1 proc get_color {obj_name} {
 2    global a_color
 3    if { [info exists a_color($obj_name)] } {
 4       return $a_color($obj_name)
 5    } else {
 6       puts "Warning: $obj_name has no color!"
 7       return "transparent"   ; # return a default color
 8    }
 9 }
10
11 proc set_color {obj_name color} {
12    global a_color
13    set a_color($obj_name) $color
14 }
```

We now access the colors of objects as follows:

example_01/apples.tcl

```
1 set_color a1 green
2 puts "a1 has color [get_color a1]"
```

The next step is to introduce some syntactic sugar: just a small improvement that makes the syntax look better, but does not really change anything fundamentally. We create the following procedure:

example_02/apples.tcl

```
1 proc a1 {command args} {
2    if { $command == "get_color" } {
3       return [get_color a1]
4    } elseif { $command == "set_color" } {
5       set_color a1 [lindex $args 0]
6    } else {
7       puts "Error: Unknown command $command"
8    }
9 }
```

Using this procedure, we can now access the color of the 'a1' object as follows:

example_02/apples.tcl

```
1 a1 set_color yellow
2 puts "a1 has color [a1 get_color]"
```

With respect to the earlier example, we basically swapped the positions of the object name with the name of the `get_color` or `set_color` procedure. Not very useful in itself, but it makes the

syntax look more like an object invocation. It looks as if we invoke the "method" `set_color` on the "object" a1.

Procedure `a1` is called an *object command* or *instance command*. Its name is the name of an object. Its first argument is the name of a method that you want to invoke on the object. The object's data is stored in a global array, in this case `a_color`, but that is hidden from the programmer by the object command.

We can now create as many objects as we want: just write a procedure like `a1` for each object, each with a different name. Sounds like a lot of work? It is. We will soon see how you can automate this. Writing a separate procedure for every object is not only tiresome; it also imposes heavy resource burdens on the application, because procedures take up space in the Tcl interpreter.

The first improvement is that we can write a single dispatcher procedure like this one:

example_03/apples.tcl

```
1 proc dispatch {obj_name command args} {
2    if { $command == "get_color" } {
3       return [get_color $obj_name]
4    } elseif { $command == "set_color" } {
5       set_color $obj_name [lindex $args 0]
6    } else {
7       puts "Error: Unknown command $command"
8    }
9 }
```

The object commands can now be written with only a single line of code:

example_03/apples.tcl

```
1 proc a1 {command args} {
2    return [eval dispatch a1 $command $args]
3 }
```

Creating a procedure of this form for each object consumes less memory, simply because the procedure is shorter. But it is still quite cumbersome to write a procedure every time you want to instantiate an object. To simplify this task, we write yet another procedure, one that creates object commands! It looks like this:

example_04/apples.tcl

```
1 proc apple {args} {
2    foreach name $args {
3       proc $name {command args} \
4          "return \[eval dispatch $name \$command \$args\]"
5    }
6 }
```

We call this procedure the *class command*, because it is like a class type that you can instantiate. Instantiating and manipulating objects is now as simple as this:

example_04/apples.tcl

```
1 apple a1 a2 a3
2 a1 set_color green
3 a2 set_color yellow
4 a3 set_color red
5 puts "a1 has color [a1 get_color]"
6 puts "a2 has color [a2 get_color]"
7 puts "a3 has color [a3 get_color]"
```

The class command creates objects of class 'apple'. Each apple has its own color, which can be accessed through the methods `get_color` and `set_color` of the class.

There are still some pieces missing in the puzzle. First of all, we now have a way of creating new objects, but we cannot delete objects yet. This leads to memory leaks, so we need to provide a procedure for deleting apples:

example_05/apples.tcl

```
proc delete_apple {args} {
   global a_color
   foreach name $args {
      unset a_color($name)   ; # Deletes the object's data
      rename $name {}   ; # Deletes the object command
   }
}
```

We can also set up the array `a_color` in such a way that `$a_color(obj)` is always initialized with a default value for every object. We do this in the class command, by setting the default color to green:

example_05/apples.tcl

```
proc apple {args} {
   global a_color
   foreach name $args {
      proc $name {command args} \
         "return \[eval dispatch $name \$command \$args\]"
      set a_color($name) green
   }
}
```

This makes the class command act like a *constructor* that sets up the default values for object attributes. In this case we picked green as the default color for apples. We now use the complete set of procedures like this:

example_05/apples.tcl

```
apple a1 a2 a3
a2 set_color yellow
a3 set_color red
puts "a1 has color [a1 get_color]"   ; # Uses default color green
puts "a2 has color [a2 get_color]"
puts "a3 has color [a3 get_color]"
delete_apple a1 a2 a3   ; # Delete the objects.
```

Summary so far

To summarize, we have followed these steps:

- Store attributes in a global array.
- Write a procedure for each 'method' of the object; this method takes the name of the object as its first argument.
- Write a dispatch procedure to call one of those methods.
- For each object, write a procedure (the *object command*) with the same name as the object. Its first argument is the method name. It calls 'dispatch', which swaps the object and method names.
- For each class, write a procedure (the *class command*) that creates the object commands automatically. The class command can also fill in default attribute values.
- For each class, write a delete procedure to reclaim resources of an object and destroy its object command.

The resulting object system is still a bit too simple, but all the basic techniques are there. You now know enough to start using object commands and class commands in Tcl. The rest of this article refines the techniques, offers a few more tips and tricks, and gives (pointers to) real-life examples where object commands are used.

More attributes

So far, apples only have a color. We will now give our `apple` class a few more attributes: a size and a price (both are integers for simplicity). These are again stored in global arrays, for example `a_size` and `a_price`. Both are indexed by the name of the object, just as for the `a_color` array we've been using so far. And again we can write get/set procedures to access these new attributes. The code is very similar to that for the color attribute, so I will not show it here.

An interesting (and frankly, much better) alternative is to use an

array for every *object*, rather than an array for every *attribute*. Tcl allows us to create a procedure and an array variable with the same name, so we can call our object command 'a1' and use an array 'a1' to store the attributes of that object. The code of all our procedures now changes slightly, because we have to index the object's array with the attribute name, rather than the attribute's array with the object name:

example_06/apples.tcl

```
# We now use the object name as an array, not the attribute name.
# This allows us to easily store and retrieve multiple attributes per object.
# To delete an object, just delete its array and its object command.

proc get_color {obj_name} {
   upvar #0 $obj_name arr
   return $arr(color)
}

proc set_color {obj_name color} {
   upvar #0 $obj_name arr
   set arr(color) $color
}

proc dispatch {obj_name command args} {
   if { $command == "get_color" } {
      return [get_color $obj_name]
   } elseif { $command == "set_color" } {
      set_color $obj_name [lindex $args 0]
   } else {
      puts "Error: Unknown command $command"
   }
}

proc apple {args} {
   foreach name $args {
      proc $name {command args} \
         "return \[eval dispatch $name \$command \$args\]"
      upvar #0 $name arr
      set arr(color) green
   }
}

proc delete_apple {args} {
   foreach name $args {
      upvar #0 $name arr
      unset arr        ; # Deletes the object's data
      rename $name {}  ; # Deletes the object command
   }
}

# Note the advantage of using an array per object:
# 'delete_apple' can just 'unset arr' instead of having to
# remove one entry in three different arrays.
```

A third alternative is to use only a single, global array, indexed by the object name *and* the attribute name. To find the color of object a1, you would have to access `$all_attributes(a1,color)`. The advantage of having only a single array to maintain, has to be weighed off against the disadvantage of having to delete several array entries when deleting an object.

Configuring the attributes

Another improvement that we can make, is to get rid of all those annoying get/set methods. We do this by introducing two new methods for each class, called `configure` and `cget`. The first gives new values to some attributes, the second reads the value of an attribute. Their names are chosen to reflect similar commands in Tk. We can implement these procedures for the `apple` class as follows:

example_07/apples.tcl

```
# Handling multiple attributes with 'configure' and 'cget'.

proc dispatch {obj_name command args} {
   upvar #0 $obj_name arr
   if { $command == "configure" || $command == "config" } {
      foreach {opt val} $args {
         if { ![regexp {^-(.+)} $opt dummy small_opt] } {
            puts "Wrong option name $opt (ignored)"
         } else {
```

```
            set arr($small_opt) $val
         }
      }

   } elseif { $command == "cget" } {
      set opt [lindex $args 0]
      if { ![regexp {^-(.+)} $opt dummy small_opt] } {
         puts "Wrong or missing option name $opt"
         return ""
      }
      return $arr($small_opt)

   } elseif { $command == "byte" } {
      puts "Taking a byte from apple $obj_name ($arr(size))"
      incr arr(size) -1
      if { $arr(size) <= 0 } {
         puts "Apple $obj_name now completely eaten!  Deleting it..."
         delete_apple $obj_name
      }

   } else {
      puts "Error: Unknown command $command"
   }
}

# We also change the implementation of the "constructor",
# so that it accepts initializing values for the attributes.
proc apple {name args} {
   proc $name {command args} \
      "return \[eval dispatch $name \$command \$args\]"

   # First set some defaults
   upvar #0 $name arr
   set arr(color) green
   set arr(size) 5
   set arr(price) 10

   # Then possibly override those defaults with user-supplied values
   if { [llength $args] > 0 } {
      eval $name configure $args
   }
}
```

The constructor now creates only a single object, but allows you to specify its attribute values right there in the constructor call. Attribute access now looks exactly as it does for Tk widgets. Compare these two fragments of code:

```
# Tk button object.
button .b -text "Hello" -command "puts world"
.b configure -command "exit"
set textvar [.b cget -text]

# Our own apple object.
apple a -color red -size 5
a configure -size 6
set clr [a cget -color]
```

Some widget libraries that are written in pure Tcl, use object commands and configure/cget methods to make the widget syntax the same as in Tk. Obviously, this technique also works for other kinds of objects.

Object persistence

We will now cover a more exotic topic: object persistence. This means that you can save an object on disk, and recover it later, in the same or in another application. The recovered object has exactly the same attribute values as the one you saved.

In languages such as C++, object persistence is quite a challenge (especially if you want to save an object on one platform, and recover it on another platform with different endianness or with a different compiler). But the flexibility of Tcl, and the fact that everything is a string, makes object persistence a piece of cake! We will save our objects in a text file, then treat that file as an Active File to read the objects back. Read more about the Active File pattern in my article on Tcl file formats.

We only need a single Tcl procedure (!) to give objects of *all* classes

the ability to make themselves persistent:

example_08/apples.tcl

```
 1 proc write_objects {classname args} {
 2    foreach name $args {
 3       upvar #0 $name arr
 4       puts "$classname $name \\"
 5       foreach attr [array names arr] {
 6          puts "    -$attr $arr($attr) \\"
 7       }
 8       puts ""
 9    }
10 }
```

The idea is to invoke this procedure as follows:

example_08/write_apples.tcl

```
1 write_objects apple a1 a2 a3
```

The implementation above shows that the procedure makes the objects a1, a2, and a3 persistent, by simply outputting a call to the constructor with the object name and all its attributes. The resulting output is stored in a data file and looks like this:

example_08/datafile.dat

```
 1 apple a1 \
 2    -price 10 \
 3    -size 5 \
 4    -color green \
 5
 6 apple a2 \
 7    -price 10 \
 8    -size 3 \
 9    -color yellow \
10
11 apple a3 \
12    -price 12 \
13    -size 5 \
14    -color red \
15
```

It is now extremely easy to read these persistent objects back from disk: just `source` the file! Tcl's `source` command executes all constructor calls in the file, creating instances with exactly the same attributes as the ones we saved earlier. Object persistence in Tcl is indeed a piece of cake.

Note that we only need a single procedure `write_objects` to persist objects of *any* class. The procedure simply outputs the class name before every object name.

Also note that we would have to add quotes around the attribute values to support values containing spaces.

Adding new classes

So far, we have worked with only a single class `apple`. If we want to add a new class to our example, we need to write a new class command and a new dispatcher procedure.

Suppose we also want to have objects of class `fridge` (in which we will want to store apples of course). We need to duplicate the effort we did on the `apple` class:

example_10/classes.tcl

```
 1 # Dispatch procedure for class 'fridge'.
 2 proc dispatch_fridge {obj_name command args} {
 3    upvar #0 $obj_name arr
 4    if { $command == "configure" || $command == "config" } {
 5       array set arr $args
 6
 7    } elseif { $command == "cget" } {
 8       return $arr([lindex $args 0])
 9
10    } elseif { $command == "open" } {
11       if { $arr(-state) == "open" } {
12          puts "Fridge $obj_name already open."
```

```
13       } else {
14          set arr(-state) "open"
15          puts "Opening fridge $obj_name..."
16       }
17
18    } elseif { $command == "close" } {
19       if { $arr(-state) == "closed" } {
20          puts "Fridge $obj_name already closed."
21       } else {
22          set arr(-state) "closed"
23          puts "Closing fridge $obj_name..."
24       }
25
26    } else {
27       puts "Error: Unknown command $command"
28    }
29 }
30
31 # Class procedure for class 'fridge'.
32 proc fridge {name args} {
33    proc $name {command args} \
34       "return \[eval dispatch_fridge $name \$command \$args\]"
35
36    # First set some defaults
37    upvar #0 $name arr
38    array set arr {-state closed -label A}
39
40    # Then possibly override those defaults with user-supplied values
41    if { [llength $args] > 0 } {
42       eval $name configure $args
43    }
44 }
```

These two procedures are almost exactly the same as for apples, and we have to repeat the work for every class we add to our script. This laborious task can be partly automated by a procedure called `class` which accepts the name of a new class, a list of its member variables, and a list of its method names. It then automatically sets up the necessary procedures such as the class command and the dispatcher procedure. The only thing we still need to implement by hand, are the methods of the class. The whole thing could be set up as follows:

example_11/classes.tcl

```
 1 proc class {classname vars methods} {
 2
 3    # Create the class command, which will allow new instances to be created.
 4    proc $classname {obj_name args} "
 5       # The class command in turn creates an object command.  Careful
 6       # with those escape characters!
 7       proc \$obj_name {command args} \
 8          \"return \\\[eval dispatch_$classname \$obj_name \\\$command \\\$args\\\]\"
 9
10       # Set variable defaults
11       upvar #0 \$obj_name arr
12       array set arr {$vars}
13
14       # Then possibly override those defaults with user-supplied values
15       if { \[llength \$args\] > 0 } {
16          eval \$obj_name configure \$args
17       }
18    "
19
20    # Create the dispatcher, which dispatches to one of the class methods
21    proc dispatch_$classname {obj_name command args} "
22       upvar #0 \$obj_name arr
23       if { \$command == \"configure\" || \$command == \"config\" } {
24          array set arr \$args
25
26       } elseif { \$command == \"cget\" } {
27          return \$arr(\[lindex \$args 0\])
28
29       } else {
30          if { \[lsearch {$methods} \$command\] != -1 } {
31             uplevel 1 ${classname}_\${command} \$obj_name \$args
32          } else {
33             puts \"Error: Unknown command \$command\"
34          }
35       }
36    "
37 }
```

The `class` procedure basically just creates two new commands for

us (a class command and a dispatcher). These are the 2 commands that we had to create by hand earlier.

The code looks pretty messy, because it contains two levels of indirection: a proc that creates a proc that creates yet another proc. This involves a bit of backslash-escape magic, which can be confusing. Richard Suchenwirth has a very nice solution to make this kind of code more readable: he creates a template with names containing a special character such as '@'. Then he replaces those names by the actual class and instance names, using `regsub`. See his page on gadgets for an example. Using this technique, our implementation becomes a lot easier to read:

example_12/classes.tcl

```
proc class {classname vars methods} {

   # Create the class command, which will allow new instances to be created.
   set template {
      proc @classname@ {obj_name args} {
         # The class command in turn creates an object command.
         # Fewer escape characters thanks to the '@' sign.
         proc $obj_name {command args} \
            "return \[eval dispatch_@classname@ $obj_name \$command \$args\]"

         # Set variable defaults
         upvar #0 $obj_name arr
         array set arr {@vars@}

         # Then possibly override those defaults with user-supplied values
         if { [llength $args] > 0 } {
            eval $obj_name configure $args
         }
      }
   }

   regsub -all @classname@ $template $classname template
   regsub -all @vars@ $template $vars template

   eval $template

   # Create the dispatcher, which dispatches to one of the class methods
   set template {
      proc dispatch_@classname@ {obj_name command args} {
         upvar #0 $obj_name arr
         if { $command == "configure" || $command == "config" } {
            array set arr $args

         } elseif { $command == "cget" } {
            return $arr([lindex $args 0])

         } else {
            if { [lsearch {@methods@} $command] != -1 } {
               uplevel 1 @classname@_${command} $obj_name $args
            } else {
               puts "Error: Unknown command $command"
            }
         }
      }
   }

   regsub -all @classname@ $template $classname template
   regsub -all @methods@ $template $methods template

   eval $template
}
```

You see that this simplifies the code. We use the '@' sign because it is not frequently used in normal Tcl code. We postpone the evaluation of `$classname` and other variables until we are out of the inner procedure body, so that the number of escape characters is reduced to almost zero.

With or without this "template" technique, we can now create our original classes `apple` and `fridge` in a much more compact way:

example_12/classes.tcl

```
# Create a class with 3 attributes and a 'byte' method.
class apple {-color green -size 5 -price 10} {byte}
proc apple_byte {self} {
   upvar #0 $self arr
   puts "Taking a byte from apple $self"
   incr arr(-size) -1
```

```
 7    if { $arr(-size) <= 0 } {
 8       puts "Apple $self now completely eaten!  Deleting it..."
 9       delete $self
10    }
11 }
12
13 # Create a class with 2 attributes and 2 methods.
14 class fridge {-state closed -label A} {open close}
15 proc fridge_open {self} {
16    upvar #0 $self arr
17    if { $arr(-state) == "open" } {
18       puts "Fridge $self already open."
19    } else {
20       set arr(-state) "open"
21       puts "Opening fridge $self..."
22    }
23 }
24
25 proc fridge_close {self} {
26    upvar #0 $self arr
27    if { $arr(-state) == "closed" } {
28       puts "Fridge $self already closed."
29    } else {
30       set arr(-state) "closed"
31       puts "Closing fridge $self..."
32    }
33 }
```

Creating new classes is indeed a lot simpler than before. We only need one line with the class "declaration" containing its attribute names, plus one proc for each of the class methods. Each method is implemented as a global proc which has the instance name as its first parameter. The parameter is called "self", which is only a convention. Any other arguments are optional.

In the implementation of each method, we access the object's array explicitly. We could make the methods less dependent on the actual implementation of the object by using `configure` and `cget` instead, for example

example_13/classes.tcl

```
1 proc fridge_close {self} {
2    if { [$self cget -state] == "closed" } {
3       puts "Fridge $self already closed."
4    } else {
5       $self configure -state "closed"
6       puts "Closing fridge $self..."
7    }
8 }
```

This depends less on our array implementation, and is perhaps slightly more readable. It is less efficient though, because the `configure` and `cget` implementations add an extra level of procedure calls with a couple of `if`s. You should probably decide for yourself which of the two ways you are going to use, depending on the importance of efficiency in your application.

Note that we can implement the `class` procedure in a slightly different way, without actually knowing in advance the list of all the variables and methods of the class. The new implementation could look like this:

example_14/classes.tcl

```
 1 # No more 'methods' argument here; 'vars' is optional.
 2 proc class {classname {vars ""}} {
 3
 4    # Create the class command, which will allow new instances to be created.
 5    set template {
 6       proc @classname@ {obj_name args} {
 7          # The class command in turn creates an object command.
 8          # Fewer escape characters thanks to the '@' sign.
 9          proc $obj_name {command args} \
10             "return \[eval dispatch_@classname@ $obj_name \$command \$args\]"
11
12          # Set variable defaults, if any
13          upvar #0 $obj_name arr
14          @set_vars@
15
16          # Then possibly override those defaults with user-supplied values
17          if { [llength $args] > 0 } {
18             eval $obj_name configure $args
```

```
19          }
20       }
21    }
22
23    set set_vars "array set arr {$vars}"
24    regsub -all @classname@ $template $classname template
25    if { $vars != "" } {
26       regsub -all @set_vars@  $template $set_vars template
27    } else {
28       regsub -all @set_vars@  $template "" template
29    }
30
31    eval $template
32
33    # Create the dispatcher, which does not check what it
34    # dispatches to.  It just follows the naming convention.
35    set template {
36       proc dispatch_@classname@ {obj_name command args} {
37          upvar #0 $obj_name arr
38          if { $command == "configure" || $command == "config" } {
39             array set arr $args
40
41          } elseif { $command == "cget" } {
42             return $arr([lindex $args 0])
43
44          } else {
45             # Here you see the naming convention explicitly: classname_command.
46             uplevel 1 @classname@_${command} $obj_name $args
47          }
48       }
49    }
50
51    regsub -all @classname@ $template $classname template
52
53    eval $template
54 }
55
56 fridge f1 -state open
57 f1 close
58
59 # Even after 'f1' is created, we can add a new method to the 'fridge'
60 # class.  'f1' automatically gets the new method.
61 proc fridge_paint {self color} {
62    puts "Painting fridge $self $color ..."
63 }
64
65 f1 paint green
```

This implementation shows that you can add new methods to an existing class, simply by implementing a new global procedure following a `classname_methodname` naming convention, with `self` as its first argument. The dispatcher procedure will find this new method even though it did not yet exist at the time the class was created. The same is true for member variables (this has silently been the case in all previous examples): just call `configure` with a new variable name, and it will end up in the object's array. Only variables specified in the `class` procedure get a default value, though; Other variables do not exist before they are first set by `configure`!

Advanced techniques

This is not the full story on Tcl objects. There are many more techniques that you can experiment with. There are also many existing object systems for Tcl that you can study, and they use many variations on the techniques we looked at above. I won't go into detail here; I just provide an overview of some of the possibilities.

Memory leaks in Tcl

Instead of storing object attributes in global arrays, you could create *local* arrays and pass them around with `upvar`. That way, these arrays disappear when they go out of scope. This stops a lot of memory leaks. You can set a trace on the array so that when it goes out of scope, you also delete the object command. Thanks to Richard Suchenwirth for this tip.

Introspection

In the spirit of Tcl, classes and instances should offer introspection. For example, you should be able to:

- Ask an instance what class it has (perhaps by simply storing "class" as an additional attribute).
- Ask the list of all classes currently available.
- Ask a class for a list of all its instances.
- Ask a class or instance for a list of all its attributes, and/or all its methods. The former can be done by simply iterating over the object's array. The latter requires `info procs` to find all the procedures that satisfy the naming convention for methods.

Persistence revisited

- Introspection allows you to save objects without explicitly giving the class name. The system can figure out the class name autonomously, by fetching the 'class' attribute from the object.
- In fact, you can now write a proc that saves *all* objects in the system, in one fell swoop.
- Combine the two into one proc: If you pass some instance names as parameters, only those instances are saved; otherwise, all instances are saved.
- Save the class commands in the same output script as the instances, so that the data can be loaded into any Tcl script without prior knowledge of the classes and their structure.

From object-based to object-oriented

Destructors, virtual functions, inheritance, delegates, properties, operator overloading, ... There are many other object-oriented features that we have not yet discussed in this article. The Tcl'ers Wiki has many cool articles that cover such techniques. Here are only a few:

- Gadgets are extremely simple, and yet powerful (they have a form of operator overloading!)
- Jean-Claude Wippler's object system stores objects as key-value records, so that they can be passed around by value. No global arrays are needed.
- Oblets use many of the techniques described in this article. The inheritance syntax uses ':' as in C++ or Java. The members in the class body are declared with a custom 'set' and 'proc'.

Namespaces

To avoid name collisions, classes and instances should be defined in namespaces. Will Duquette's article covers this in detail.

Reference links

Many object-oriented extensions for Tcl have been implemented, and since version 8.6, Tcl even has a built-in object system. Find out more about all these object systems at the following pages:

- Since version 8.6, Tcl has its own official object system, which is very flexible and allows many other object mechanisms to be implemented on top of it. See Tcl8.6 objects. It was inspired by Snit and XOtcl, among others.
- SWIG is an excellent tool for making C++ objects available in Tcl. You can combine SWIG with some of the techniques in this article to provide an object-oriented Tcl interface for your C++ classes. Obviously, this requires compilation of the C++ classes and the generated wrappers.
- [Incr Tcl] is one of the earliest object-oriented extensions of Tcl. It requires compilation. Classes are static (cannot be altered once declared). Each class has its own namespace, but [Incr Tcl] does not use the standard Tcl namespace mechanism. Supports inheritance, *ensembles*, and reflection. Blends very well with Tk.
- OTcl is an object-oriented extension of the Tcl language. It requires compilation.
- XOTcl is a more advanced object-oriented extension of Tcl

which also requires recompilation. It served as one of the sources of inspiration for the official Tcl8.6 object system. It offers many advanced features such as meta-classes, multiple inheritance, instance-specific methods, nested classes, assertions, mixins, filters, ...
- Stooop is entirely written in Tcl, so no extra compilation or linking is required.
- Snit is also written in pure Tcl (no compilation required). Its purpose is mainly to glue existing object systems together, rather than implementing new classes. To achieve this, it uses *delegation* instead of inheritance. This dynamic flexibility was one of the sources of inspiration for the official Tcl8.6 object system.
- Toot is a very simple and elegant mechanism for turning existing Tcl commands into "classes". Objects are passed by value rather than by reference, so the attribute values are contained inside the object's string representation instead of a global array.
- Richard Suchenwirth pointed me to the Lightweight Object System for Tcl ("LOST") on the Tcl'ers wiki, and the very elegant Tcl Gadgets on which it is based. Both are very light-weight but still quite powerful.
- Will's Guide to Creating Object Commands is another article on how to implement your own object commands in Tcl. It covers more advanced topics such as dealing with object commands in namespaces.
- Cetus Links maintains a page *full* of links about Tcl; some of them are about objects too. Unfortunately, many of the links are out of date, so your mileage may vary.

Many thanks to everybody who read the first drafts of this article and helped me correct some mistakes and make many improvements. Special thanks to Richard Suchenwirth, Jean-Luc Fontaine, and Bob Techentin for their technical insights.