

Selfish 1.0 — A Self-like object extension for Tcl

KRISTOFFER LAWSON, <mailto:setok@altparty.org>

6 October 2006

1 Overview

Selfish is yet another package for object oriented programming in Tcl, but with quite a difference from the others offered. It is based on the language Self[1] and the concepts presented in it. The aspects of Selfish which separate it from other extensions are:

- There are no classes or types. Any object can inherit from any other to facilitate shared behaviour.
- As there are no classes, there is no object instantiation. Instead, the biological notion of cloning is used. Cloned objects are generally called prototypes and when they are cloned their inheritance and all information about them is copied to a new object. Any object can be cloned.
- There is no separation between methods and fields. There are only slots which can return values. They may or may not do more than return a value. It is transparent to the user.
- All relations and slots are dynamic and can be modified at any time.
- Written in pure Tcl.

It is this simplicity which makes a system such as this so powerful, in a similar fashion to Tcl itself. It also means it is easy to understand in whole and lends to a straightforward implementation. This form does not always obviously map to the class-object models taught to people and can take some getting used to. However, the belief behind this project is that this leads to a power and conceptual neatness that it is worth exploring. If wanted, a class-object model can be built on top of a prototype object system, but it may not always be the best solution to design problems.

2 Objects and slots

Each object is effectively responsible for itself. They act independently of each other in a way which is not as stark in class-based systems. Each object has a series of slots. Messages passed to these objects are matched with a named slot of the object (roughly

the equivalent of methods). An object can inherit from a list of any other objects. If the `parents*` slot is set and a message cannot be mapped to a slot in the object, the message will be searched for in the parents, starting with the first in the list (and up its inheritance chain) and then onto the next, and so forth. There the code attached to the slot will be evoked in the context of the original receiver.

In addition to inheritance, any Selfish object can be cloned. The state, inheritance and indeed all slots will be copied to a new object which is an exact replica of the original. This can be vaguely compared to instantiation in normal class-based systems.

Here is how an object in the `myObject` variable would be cloned:

```
set newOb [$myObject clone]
```

If we wish to add a slot to `newOb` we can do the following:

```
selfish::slot $newOb hello {} {  
  puts "Hello, world"  
}
```

This creates a new slot called `hello` which does not take any arguments. When called, it displays `Hello, world` to the standard output. It can be called as follows:

```
$newOb hello
```

As a shortcut, if one wishes to create a slot that simply returns a value (and allows to change that value) we can use the following:

```
selfish::valueSlot $newOb x 10
```

This creates a slot called `x` which initially returns the value 10. This slot can be called to assign new values to itself.

```
$newOb x 20
```

Naturally it can then be accessed to receive the value.

```
$newOb x
```

The above will return the value 20.

The `selfish::valueSlot` command's last argument (the value) is optional. If one is not given, a slot will be created but without a value and an error will be thrown if it is accessed without first assigning a value to it.

Note that the `selfish::valueSlot` command is just shorthand for writing code which would do something similar with that slot. As everything in Selfish is dynamic, this slot can be replaced at any time with a block of code which would do something before returning a value. This will be completely transparent to anything using this slot.

To delete a slot, use the `selfish::deleteSlot <ob> <slot>` command. It only applies to slots created for `ob` directly, not to any inherited.

3 Code in slots

Blocks of code attached to slots act like the bodies of procedures in Tcl. In addition to all the normal functionalities of Tcl they can use the `me` command to receive a handle to the object currently receiving the message. Note that this can be different from the object in which the slot is defined, if the original target object inherited from the object in which the slot was defined. Code bodies can also access their slots directly as commands but this bypasses the normal slot lookup: it accesses the slot of the object where the code body was defined. F.ex. say we have an object in the `Point` variable and another in the `NewPoint` variable:

```
set Point [baseObject clone]
selfish::slot $Point print {} {
    puts "x: [[me] x]"
    puts "y: [[me] y]"
}
selfish::valueSlot $Point x 10
selfish::valueSlot $Point y 20
set NewPoint [baseObject clone]
selfish::valueSlot $NewPoint parents* $Point
selfish::valueSlot $NewPoint x 5
selfish::valueSlot $NewPoint y 5

$NewPoint print
```

In the above case the following would be printed:

```
x: 5
y: 5
```

But if the `print` slot in `$Point` had been defined as follows:

```
selfish::slot $Point print {} {
    puts "x: [x]"
    puts "y: [y]"
}
```

The following would be printed instead:

```
x: 10
y: 20
```

Often it is probably a good idea to use the version with `me`, unless there is a specific reason not to. Note also that direct-access is exactly that. If the slot does not exist for the object where the slot code resides its parents will not be checked.

4 Bugs

Lots of 'em?

References

- [1] David Ungar, Randall B. Smith: *Self: The Power of Simplicity*. <http://research.sun.com/self/>
- [2] The Tool Command Language, <http://www.tcl.tk/>