stooop

(Simple Tcl Only Object Oriented Programming)

Stooop is an extension to the great Tcl language written in Tcl itself. The object oriented features of stooop are modeled after the C++ programming language while following the Tcl language philosophy.

### Contents

* [About this document](#_gjdgxs)
* [Introduction](#_30j0zll)
* [Simple example](#_1fob9te)
* [Coding conventions](#_3znysh7)
  + [Class definition](#_2et92p0)
  + [Member procedures](#_tyjcwt)
    - [Constructor](#_3dy6vkm)
    - [Destructor](#_1t3h5sf)
    - [Non static](#_4d34og8)
    - [Static](#_2s8eyo1)
    - [Copy constructor](#_17dp8vu)
  + [Member data](#_3rdcrjn)
    - [Non static](#_26in1rg)
    - [Static](#_lnxbz9)
* [Commands](#_35nkun2)
  + [class](#_1ksv4uv)
  + [new](#_44sinio)
  + [delete](#_2jxsxqh)
  + [virtual](#_z337ya)
  + [classof](#_3j2qqm3)
* [Package](#_1y810tw)
  + [Installation](#_4i7ojhp)
  + [Creation](#_2xcytpi)
* [Examples](#_1ci93xb)
  + [Parallel with C++](#_3whwml4)
  + [Graphical demonstration](#_2bn6wsx)
  + [Widget class](#_qsh70q)
  + [Member array](#_3as4poj)
* [Utility classes](#_1pxezwc)
  + [switched](#_49x2ik5)
* [Debugging](#_2p2csry)
  + [member check](#_147n2zr)
    - [procedure](#_3o7alnk)
    - [data](#_23ckvvd)
  + [member trace](#_ihv636)
    - [procedure](#_32hioqz)
    - [data](#_1hmsyys)
  + [objects](#_41mghml)
    - [printing](#_2grqrue)
    - [recording](#_vx1227)
    - [reporting](#_3fwokq0)
* [Notes](#_1v1yuxt)
  + [On design choices](#_4f1mdlm)
  + [On implementation](#_2u6wntf)
* [Miscellaneous information](#_19c6y18)

### About this document

This document contains general information, reference information and many examples designed to help the programmer understand and use the stooop extension (version 4.1.1 and above).

A working knowledge of object oriented programming techniques and a related programming language (C++, Java, ...) significantly helps understand this document.

### Introduction

After some time writing Tcl/Tk code, I felt that I needed a way to improve the structure of my code, and why not use an object oriented approach, since I knew (but does anybody really? :-) C++. As I have used Tcl quite extensively in several commercial applications running on different operating systems and hardware, I decided to use a strict Tcl implementation for my object oriented extension. Consequently, stooop is compatible with all the Tcl ports (UNIX, Windows, MacIntosh).

Great care was taken so that this extension would no adverse impact on performance. Furthermore, designing your code in an object oriented should improve its performance, by focusing on well written pieces of reusable code.

Stooop only introduces a few new commands: [class](#_1ksv4uv), [new](#_44sinio), [delete](#_2jxsxqh), [virtual](#_z337ya) and [classof](#_3j2qqm3). Along with a few coding conventions, that is basically all you need to know to use stooop. Stooop is meant to be as simple to use as possible.

Starting with stooop version 3.2, nested classes are supported (see [class](#_1ksv4uv)), whereas version 3.3 and above support procedure and data members checking as well as tracing (see [debugging](#_2p2csry)).

Tcl version 8.2 and above supports the empty name array syntax, as in:

set (m) 0 ;# set member m of array {} to 0  
set n $(m) ;# which actually sets n to 0

This feature greatly simplifies class member manipulation in stooop classes and significantly improves performance. Stooop version 4.0 and above also uses this feature internally for further improvements, without sacrificing backward compatibility: code written against stooop versions 3.7 and below still works with stooop version 4.0 and above, but can be gradually moved to the simpler syntax when convenient.

Stooop 4.1 and above will only work out of the box with Tcl 8.3 and above.

### Simple example

Let us start with a code sample that will give you some feeling on how stooop works:

package require stooop 4 ;# load stooop package  
namespace import stooop::\* ;# and import class, new, ... commands

class shape { ;# base class definition  
 proc shape {this x y} { ;# base class constructor  
 set ($this,x) $x ;# data member initialization  
 set ($this,y) $y  
 }  
 proc ~shape {this} {} ;# base class destructor  
 # pure virtual draw: must be implemented in derived classes  
 virtual proc draw {this}  
 virtual proc rotate {this angle} {} ;# do nothing by default  
}  
proc shape::move {this x y} { ;# external member procedure definition  
 set ($this,x) $x  
 set ($this,y) $y  
 draw $this ;# shape::draw invokes derived class implementation  
}  
  
class triangle { ;# class definition  
 proc triangle {this x y} shape {$x $y} { ;# derived from shape  
 # triangle constructor implementation  
 }  
 proc ~triangle {this} {}  
 proc draw {this} {  
 # triangle specific implementation  
 }  
 proc rotate {this angle} {  
 # triangle specific implementation  
 }  
}  
  
class circle {} ;# empty class definition, procedures are defined outside  
proc circle::circle {this x y} shape {$x $y} { ;# derived from shape  
 # circle constructor implementation  
}  
proc circle::~circle {this} {}  
proc circle::draw {this} {  
 # circle specific implementation  
}  
# circle::rotate procedure is a noop, no need to overload  
  
lappend shapes [new circle 20 20] [new triangle 80 20]  
foreach object $shapes {  
 shape::draw $object  
 shape::rotate $object 45  
}  
eval delete $shapes

### Coding conventions

I have tried to make stooop Tcl code look like C++ code. There are exceptions of course.

#### Class definition

The syntax is very simple:

class className { ...

The member procedures are then defined, inside or outside the class definition (see below). Note that the base classes if any are defined within the constructor declaration where they are required for eventually passing constructor parameters, not in the actual class declaration where they would then be redundant.

As a class is a namespace, it is just as easy to nest classes as it is namespaces.

#### Member procedures

They can be defined inside or outside their class definition. When defined inside the class definition, the class name qualifier (*shape::* for example) before the procedure name must be omitted (a class is a Tcl namespace). When defined outside the class definition, the class name qualifier must be present (same reason). You may notice that the class definition and the related member procedures look very much like the Tcl *namespace* feature: it is because classes are indeed namespaces with a few more features added to support object orientation.

Member procedures are named as in C++ (for example, the *rotate* procedure of the class *shape* is referred to as *shape::rotate* in the global namespace). They are defined using the Tcl *proc* command, which is redefined by stooop in order to do some specific additional processing. Of course, global level and other namespaces procedures are not affected by stooop.

##### Constructor

A constructor is used to initialize an object of its class. The constructor is invoked by the [new](#_44sinio) operator when an object of the class is created (instanciated in OO terms). The constructor is named as in C++ (for example, the *shape* constructor fully qualified name is *shape::shape*).

The constructor always takes the object identifier (a unique value generated by the command new) as the first parameter, plus eventually additional parameters as in the normal Tcl proc command. Arguments with default values are allowed, and so are variable number of arguments (see below). In all cases, the first parameter must be named **this**.

***Note****: the object identifier is a unique integer value which is internally incremented by stooop each time a new object is created. Consequently, the greater the object identifier, the younger the object.*

Sample code of a constructor of a simple class with no base class:

class shape {  
 proc shape {this x y} {  
 # implementation here  
 }  
}

If a class is derived from one or more base classes, the derived class constructor defines the base classes and their constructor arguments before the actual body of the constructor.

***Note****: base classes are not defined at the class command level, because it would be redundant with the constructor definition, which is mandatory.*

The derived class constructor parameters are followed by "base class names / constructor arguments" pairs. For each base class, there must be a corresponding list of constructor arguments to be used when the object is constructed when the new operator is invoked with the derived class name as argument.

Sample code for a class constructor with a single base class:

class circle {}  
proc circle::circle {this x y} shape {$x $y} {  
 # circle constructor implementation  
}

Sample code for a class constructor with multiple base classes:

class hydroplane {  
 proc hydroplane {this wingspan length} plane {  
 $wingspan $length  
 } boat {  
 $length  
 {  
 # constructor implementation  
 }  
}

The base class constructor arguments must be prefixed with dollar signs since they will be evaluated at the time the object is constructed, right before the base class constructor is invoked. This technique allows, as in C++, some actual processing to be done on the base class arguments at construction time. The **this** argument to the base class constructor must not be specified for it is automatically generated by stooop.

Sample code for a derived class constructor with base class constructor arguments processing:

class circle {  
 proc circle {this x y} shape {  
 [expr round($x)] [expr round($y)]  
 } {  
 # constructor implementation  
 }  
}

The base class(es) constructor(s) is(are) automatically invoked before the derived class constructor body is evaluated. Thus layered object construction occurs in the same order as in C++.

Variable length arguments are a special case and depend on both the derived class constructor arguments and those of the base class.

If both derived and base class constructors take a variable number of arguments (through the *args* special argument (see Tcl proc manual page)), the base class constructor will also see the variable arguments part as separate arguments. In other words, the following works as expected:

class base {}  
proc base::base {this parameter args} {  
 array set options $args  
}  
class derived {}  
proc derived::derived {this parameter args} base {  
 $parameter $args  
} {}  
new derived someData -option value -otherOption otherValue

Actually, if you want to get fancy, to allow some processing on the derived class constructor variable arguments, the last element (and only the last) of the derived class constructor arguments is considered variable if it contains the string *$args*. For example:

class base {  
 proc base {this parameter args} {  
 array set options $args  
 }  
}  
class derived {  
 proc derived {this parameter args} base {  
 $parameter [process $args]  
 } {}  
 proc process {arguments} {  
 # do some processing on arguments list  
 return $arguments  
 }  
}  
new derived someData -option value -otherOption otherValue

##### Destructor

The destructor is used to clean up an object before it is removed from memory. The destructor is invoked by the [delete](#_2jxsxqh) operator when an object of the class is deleted. The destructor is named as in C++ (for example, the shape constructor fully qualified name is *shape::~shape*).

The destructor always takes the object identifier (a unique value previously generated and returned by the operator new) as the only parameter, which must be named **this**.

The base class(es) destructor(s) is(are) invoked at the end of the derived class destructor body. Thus layered object destruction occurs in the same order as in C++.

Sample code of a class destructor:

class shape {  
 proc ~shape {this} {  
 # implementation here  
 }  
}

Contrary to C++, a destructor cannot (nor does it need to) be [virtual](#_z337ya). Even if it does nothing, a destructor **must** always be defined.

##### Non static

A *non static* member procedure performs some action on an object of a class. The member procedure is named as a member function in C++ (for example, the shape class move member procedure is known as *shape::move* in the Tcl global namespace).

The member procedure always takes the object identifier (a unique value generated and returned by the operator new) as the first parameter, plus eventually additional parameters as in the normal Tcl proc command. Arguments with default values are allowed, and so are variable number of arguments. In all cases, the first parameter must be named **this**.

Sample code of a member procedure:

proc shape::move {this x y} {  
 set ($this,x) $x  
 set ($this,y) $y  
 draw $this ;# invoke another member procedure  
}

A non static member procedure may be a [virtual](#_z337ya) procedure.

##### Static

A *static* member procedure performs some action independently of the individual objects of a class. The member procedure is named as a member function in C++ (for example, the shape class add static member procedure is defined as *shape::add* outside its class definition, *add* inside).

However, with stooop, there is no static specifier: a member procedure is considered static if its first parameter is not named **this**. Arguments to the procedure are allowed as in the normal Tcl proc command. Arguments with default values are also allowed, and so are variable number of arguments.

Sample code of a static member procedure:

proc shape::add {newShape} {  
 # append new shape to global list of shape  
 lappend ($shapes) $newShape  
}

Often, static member procedures access static member data (see [Static Member Data](#_lnxbz9)).

A static member procedure may not be a virtual procedure.

##### Copy constructor

***Note****: if you never create objects by copying (which is generally the case), you can skip this section.*

Let us start by making it clear that stooop generates a default copy constructor whenever a class main constructor is defined. This default copy constructor just performs a simple per data member copy, as does C++.

The user defined class copy constructor is optional as in C++. If it exists, it will be invoked (instead of the default copy constructor) when the operator [new](#_44sinio) is invoked on an object of the class or a derived class.

The copy constructor takes 2 arguments: the *this* object identifier used to initialize the data members of the object to be copied to, and the *copy* identifier of the object to be copied from, as in:

proc plane::plane {this copy} {  
 set ($this,wingspan) $($copy,wingspan)  
 set ($this,length) $($copy,length)  
 set ($this,engine) [new $($copy,engine)]  
}

As in regular member procedures, the first parameter name must be **this**, whereas the second parameter must be named **copy** to differentiate from the class constructor. In other words, the copy constructor always takes 2 and only 2 arguments (named this and copy).

The copy constructor must be defined when the default behavior (straightforward data members copy) (see the [new operator](#_44sinio)) is not sufficient, as in the example above. It is most often used when the class object contains sub objects. As in C++ when sub objects are referenced through pointers, only the sub object identifiers (see them as pointers) are copied when an object is copied, not the objects they point to. It is then necessary to define a copy procedure that will actually create new sub objects instead of just defaulting to copying identifiers.

If the class has one or more base classes, then the copy constructor must pass arguments to the base class(es) constructor(s), just as the main constructor does, as in the following example:

class ship {  
 proc ship {this length} {}  
}  
class carrier {}  
proc carrier::carrier {this length} ship {$length} {}  
proc carrier::carrier {this copy} ship {  
 $ship::($copy,length)  
} {  
 set ship::($this,planes) {}  
 foreach plane $ship($copy,planes) { ;# copy all the planes  
 lappend ship($this,planes) [new $plane]  
 }  
}

The stooop library checks that the copy constructor properly initializes the base class(es) through its(their) constructor(s) by using the regular constructor as reference. Obviously and consequently, stooop also checks that the regular constructor is defined prior to the copy constructor.

If you use [member arrays](#_3as4poj), you must copy them within the copy constructor, as they are not automatically handled by stooop, which only knows [member data](#_3rdcrjn) in the automatically generated default copy constructor.

#### Member data

All class and object data is stored in an associative array local to the class namespace (remember, a class is actually a namespace). The array name is empty, and the corresponding Tcl variable declaration is automatically inserted within class namespace and procedures (but you do not need to worry about this transparent operation).

Sample code:

class shape {}  
proc shape::shape {this x y} {  
 # set a few members of the class namespace empty named array  
 set ($this,x) $x  
 set ($this,y) $y  
 # now read them  
 puts "coordinates: $($this,x), $($this,y)"  
}

In order to access other classes data, whether they are base classes or not, a fully qualified name is always required, whereas no special declaration (global, variable, ...) is required.

Sample code:

proc circle::circle {this x y diameter} shape {$x $y} {  
 set ($this,diameter) $diameter  
 puts "coordinates: $shape::($this,x), $shape::($this,y)"  
}

##### Non static

Non static data is indexed within the class array by prepending the object identifier (return value of the *new* operator) to the actual member name. A comma is used to separate the identifier and the member name.

Much as an object pointer in C++ is unique, the object identifier in *stooop* is also unique. Access to any base class data is thus possible by directly indexing the base class array.

Sample code:

proc shape::shape {this x y} {  
 set ($this,x) $x  
 set ($this,y) $y  
}  
proc circle::circle {this x y diameter} shape {$x $y} {  
 set ($this,diameter) $diameter  
}  
proc circle::print {this} {  
 puts "circle $this data:"  
 puts "diameter: $($this,diameter)"  
 puts "coordinates: $shape::($this,x), $shape::($this,y)"  
}

##### Static

*Static* (as in C++) data members are simply stored without prepending the object identifier to the member name, as in:

proc shape::register {newShape} {  
 lappend (list) $newShape ;# append new shape to global list of shapes  
}

### Commands

Only 4 new commands [class](#_1ksv4uv), [new](#_44sinio), [delete](#_2jxsxqh) and [virtual](#_z337ya) need to be known in order to use *stooop*. Furthermore, their meaning should be obvious to C++ programmers. There is also a [classof](#_3j2qqm3) command that you can use if you need RTTI (runtime type identification).

#### class

The **class** command introduces a new class declaration.

A class is also a namespace although you do not need to worry about it, but it does have some nice side effects. The following code works as expected:

class shape {  
 set (list) {} ;# initialize list of shapes, a static data member  
 proc shape {this x y} {  
 lappend (list) $this ;# keep track of new shapes  
 }  
 ...  
}

This works because all data for the class (static and non static) is held in the empty named array, which the class command declares as a variable (see the corresponding Tcl command) for the class namespace and within every member procedure.

Starting with version 3.2, nested classes are allowed, which makes the following code possible:

class car {  
 proc car {this manufacturer type} {  
 set ($this,wheels) [list\  
 [new wheel 18] [new wheel 18] [new wheel 18] [new wheel 18]\  
 ]  
 ...  
 }  
 ...  
 class part {  
 ...  
 }  
 class wheel {  
 proc wheel {this diameter} car::part {} {  
 set ($this,diameter) $diameter  
 ...  
 }  
 proc print {this} {  
 puts "wheel of $($this,diameter) diameter"  
 }  
 ...  
 }  
}

There is quite a lot to say about the example above.

First, why would I use a nested class? Because it is cleaner that creating *carPart* and *carWheel* classes and saves on global namespace pollution.

Second, why does "*new wheel*" work from inside the car constructor? Because it invokes the *wheel::wheel* constructor, visible from the car namespace.

Third, why can't I simply derive wheel from *part* instead of *car::part*? Well, you must fully qualify the class that you derive from because the *part::part* constructor is not visible from within the wheel namespace.

Whenever you have a problem with nested classes, think in terms of namespaces, as classes are indeed namespaces (it should be clear to you by now :-).

#### new

The *new* operator is used to create an object of a class, either by explicit construction, or by copying an existing object.

When explicitly creating an object, the first argument is the class name and is followed by the arguments needed by the class constructor. New when invoked generates a unique identifier for the object to be created. This identifier is the value of the **this** parameter, first argument to the class constructor, which is invoked by new.

Sample code:

proc shape::shape {this x y} {  
 set ($this,x) $x  
 set ($this,y) $y  
}  
set object [**new** shape 100 50]

new generates a new object identifier, say 1234. shape constructor is then called, as in:

shape::shape 1234 100 50

If the class is derived from one or more base classes, the base class(es) constructor(s) will be automatically called in the proper order, as in:

proc hydroplane::hydroplane {this wingspan length} plane {  
 $wingspan $length  
} boat {  
 $length  
} {}  
set object [**new** hydroplane 10 7]

new generates a new object identifier, say 1234, plane constructor is called, as in:

plane::plane 1234 10 7

then boat constructor is called, as in:

boat::boat 1234 7

finally hydroplane constructor is called, as in:

hydroplane::hydroplane 1234 10 7

The new operator can also be used to copy objects when an object identifier is its only argument. A new object of the same class is then created, copy of the original object.

An object is copied by copying all its data members (but not including [member arrays](#_3as4poj)) starting from the base class layers. If the copy constructor procedure exists for any class layer, it is invoked by the *new* operator **instead** of the default data member copy procedure (see the [copy constructor](#_17dp8vu) section for examples).

Sample code:

set plane [new plane 100 57 RollsRoyce]  
set planes [list $plane [new $plane] [new $plane]]

#### delete operator

The *delete* operator is used to delete one or several objects. It takes one or more object identifiers as argument(s). Each object identifier is the value returned by *new* when the object was created. Delete invokes the class destructor for each object to be deleted.

Sample code:

proc shape::shape {this x y} {}  
proc shape::~shape {this} {  
  
proc triangle::triangle {this x y} shape {$x $y} {}  
proc triangle::~triangle {this} {}  
  
proc circle::circle {this x y} shape {$x $y} {}  
proc circle::~circle {this} {}  
  
set circle [new circle 100 50]  
set triangle [new triangle 200 50]  
**delete** $circle $triangle

circle identifier is set to, say 1234, triangle identifier is set to, say 1235. delete circle object first, circle destructor is invoked, as in:

circle::~circle 1234

then shape destructor is invoked, as in:

shape::~shape 1234

then delete triangle object...

For each object class, if it is derived from one or more base classes, the base class(es) destructor(s) are automatically called in reverse order of the construction order for base class(es) constructor(s), as in C++.

If an error occurs during the deletion process, an error is returned and the remaining delete argument objects are left undeleted.

#### virtual specifier

The *virtual* specifier may be used on member procedures to achieve dynamic binding. A procedure in a base class can then be redefined (overloaded) in the derived class(es).

If the base class procedure is invoked on an object, it is actually the derived class procedure which is invoked, if it exists**\***. If the base class procedure has no body, then it is considered to be a pure virtual and the derived class procedure is always invoked.

**\*** *as in C++, virtual procedures invoked from the base class constructor result in the base class procedure being invoked, not the derived class procedure. In stooop, an error always occurs when pure virtual procedures are invoked from the base class constructor (whereas in C++, behavior is undefined).*

*\* but there is a small difference with C++ behavior: for a virtual procedure to keep his nature down the derived classes hierarchy, it must be defined at each derivation level. That is, the virtual nature may be lost, for example in indirectly derived classes (see example below). Fixing this difference would have a non negligible impact on performance for a small gain in usefulness.*

Sample code:

class shape {  
 proc shape {this x y} {}  
 # pure virtual draw: must be implemented in derived classes  
 **virtual** proc draw {this}  
 **virtual** proc transform {this x y} {  
 # base implementation  
 }  
}  
class circle {}  
proc circle::circle {this x y} shape {$x $y} {}  
proc circle::draw {this} {  
 # circle specific implementation  
}  
proc circle::transform {this} {  
 shape::\_transform $this ;# use base class implementation  
 # add circle specific implementation here...  
}  
  
lappend shapes [new circle 100 50]  
foreach object $shapes {  
 # draw and move each shape  
 shape::draw $object  
 shape::move $object 20 10  
}

It is possible to invoke a virtual procedure as a non virtual one, which is handy when the derived class procedure must use the base class procedure. In this case, directly invoking the virtual base class procedure would result in an infinite loop. The non virtual base class procedure name is simply the virtual procedure name with 1 underscore ( \_ ) prepended to the member procedure name (see sample code above).

Constructors, destructors and static member procedures cannot be *virtual*.

Sample code highlighting small difference with C++:

class A {  
 proc A {this} {}  
 proc ~A {this} {}  
 **virtual** proc p {this} {puts A}  
}  
class B {  
 proc B {this} A {} {}  
 proc ~B {this} {}  
}  
class C {  
 proc C {this} B {}{}  
 proc ~C {this} {}  
 **virtual** proc p {this} {puts C}  
}  
  
set object [new C]  
A::p $object ;# prints "A" instead of "C"  
  
**virtual** proc B::p {this} {puts B}  
  
A::p $object ;# now prints "C"

#### classof operator

The *classof* command takes an object identifier as its only argument. It returns the class name of the object (name used with new when the object was created). Thus if needed, RTTI (runtime type identification) can be used as in C++, for example to create "virtual constructors".

proc shape::shape {this x y} {}  
set id [new shape 100 50]  
puts "object $id class name is [**classof** $id]"

### Package

For general information about the Tcl (version 7.5 and above) *package* facilities, refer to the corresponding manual pages.

#### Installation

A *pkgIndex.tcl* file is provided so that stooop and the [switched](#_49x2ik5) class can be installed as a package. Refer to the [INSTALL](http://docs.google.com/INSTALL) file for complete instructions and examples.

#### Creation

Before creating a package that uses stooop, stooop itself **must** be installed as a package (see above).

If you have created an object oriented library which uses stooop, you may want to make a package out of it. Unfortunately, using the default Tcl *pkg\_mkIndex* procedure (see the corresponding manual page) will not work.

Stooop checks that a base class constructor is defined before any of its derived classes constructors. Thus, the first time a derived class object is created, the base class definition file must be sourced to avoid an error. The specific *mkpkgidx.tcl* utility handles such cases and must be used to create stooop compatible package index files.

Let us suppose that you created a library with different classes spread in different source files: *lib1.tcl*, *lib2.tcl*, ..., *libn.tcl*. Of course, some of these files may contain base classes for derived classes in other files. As recommended in the pkg\_mkIndex Tcl manual page, each source file should contain a **package provide** command (although it seems to be needed only in the first source file). For example, if your package name is *foo* and the version *1.2*, the following line should appear around the beginning of each of the libn.tcl files:

package provide foo 1.2

It is now time to create the *pkgIndex.tcl* file, which is the missing piece for your foo package, with the *mkpkgidx.tcl* utility. The syntax is:

interpreter mkpkgidx.tcl packageName file [file ...]

where *interpreter* can be either tclsh or wish depending on whether your library uses Tk or not.

Enter the following command in the directory where the libn.tcl files reside:

$ tclsh mkpkgidx.tcl foo lib1.tcl lib2.tcl ... libn.tcl

or

$ wish mkpkgidx.tcl foo lib1.tcl lib2.tcl ... libn.tcl

For this to work, the source files must be ordered so that base classes are defined before any of their derived classes. If not the case, such errors are automatically caught by the stooop package index utility, which uses the stooop library itself.

If your package requires other packages and you do not wish to add the corresponding "package require" to your package source files, use the -p option, as in:

$ wish mkpkgidx.tcl -p ppp.1 -p qqq -p rrr.3.2 foo lib1.tcl lib2.tcl ... libn.tcl

Note that you may use as many -p option / value pairs as needed. Each package name is optionally followed by its version number after a . separator. If specified, the version number follows the same rules as the "package require" Tcl command. Of course, each specified package must be installed and working properly before attempting the mkpkgidx.tcl utility.

Once this is done, a pkgIndex.tcl file will have been created in the current directory. To install the package, enter for example:

$ mkdir /usr/local/lib/foo  
$ cp pkgIndex.tcl lib1.tcl lib2.tcl ... libn.tcl /usr/local/lib/foo/

You may of course install the foo package in another directory: refer to the pkg\_mkIndex Tcl manual page for further instructions.

Now in order to use your newly created packaged library in your application, just insert the following 3 lines at the beginning of the application source file:

package require stooop  
namespace import stooop::\*  
package require foo 1.2

### Examples

#### Parallel with C++

For C++ programmers, this simple parallel with C++ may make things easier to understand. First without virtual functions:

**C++:**

class className {  
 public:  
 someType someMember;  
 className(someType parameter)  
 {  
 someMember = parameter;  
 }  
 className(className &object)  
 {  
 ...  
 }  
 doSomething(someType parameter);  
 ~className(void) {  
 ...  
 }  
 };  
 someType className::doSomething(someType parameter)  
 {  
 ...  
 }  
 someType someValue;  
 className \*someObject = new className(someValue);  
 someType a = someObject->doSomething(someValue);  
 someType b = someObject->someMember;  
 className \*otherObject = new className(\*someObject);  
 delete someObject;

**(stooop'd up :) Tcl:**

class className {  
 proc className {this parameter} {  
 # new keeps track of object identifiers and passes a unique one  
 # to the constructor  
 set ($this,someMember) $parameter  
 }  
 proc className {this copy} {  
 # copy constructor  
 ...  
 }  
 proc ~className {this} {  
 # delete invokes this procedure then takes care of deallocating  
 # className array data members for this object identifier  
 ...  
 }  
 }  
 proc className::doSomething {this parameter} {  
 ...  
 }  
 set someObject [new className $someValue]  
 # invokes className::className  
 set a [className::doSomething $someObject $someValue]  
 set b $className::($someObject,someMember)  
 # copy object, className copy constructor is invoked  
 set otherObject [new $someObject]  
 delete $someObject  
 # invokes className::~className then frees members data

Now, with virtual functions:

**C++:**

class baseClassName {  
 public:  
 virtual void doSomething(someType) {}  
 baseClassName(void) {}  
 virtual ~baseClassName(void) {}  
 };  
 class derivedClassName: public baseClassName {  
 public:  
 void doSomething(someType);  
 derivedClassName(void) {}  
 ~derivedClassName(void) {}  
 };  
 void derivedClassName::doSomething(someType parameter)  
 {  
 ...  
 }  
 derivedClassName \*someObject = new derivedClassName();  
 someObject->doSomething(someValue); // derived function actually called  
 cout << typeid(\*someObject).name() << endl; // print object class name  
 delete someObject; // derived destructor called first

**Tcl with stooop:**

class baseClassName {  
 proc baseClassName {this} {  
 # sub-class is remembered so that virtual procedures may be used  
 ...  
 }  
 proc ~baseClassName {this} {  
 # cleanup at base level here...  
 }  
 virtual proc doSomething {this parameter} {  
 # derived class procedure with the same name may be invoked  
 # any code that follows is not executed if this procedure is  
 # overloaded in derived class  
 ...  
 }  
 }  
 class derivedClassName {  
 proc derivedClassName {this} baseClassName {} {  
 # base class constructor is automatically invoked  
 ...  
 }  
 proc ~derivedClassName {this} {  
 # cleanup at derived level here...  
 # base class destructor is automatically invoked  
 }  
 }  
 proc derivedClassName::doSomething {this parameter} {  
 # code that follows is executed when base class procedure is called  
 ...  
 }  
 set someObject [new derivedClassName]  
 # access object as base object, derived class procedure is actually invoked  
 baseClassName::doSomething $someObject $someValue  
 puts [classof $someObject] ;# print object class name  
 delete $someObject ;# delete object

#### Graphical demonstration

A demonstration using the Composite pattern from the great book Design Patterns, Elements of Reusable Object Oriented Software, which I heartily recommend.

The pattern is used to define a class hierarchy of the graphic base class, picture, oval and rectangle derived classes. A picture object can contain any number of other graphic objects, thus allowing graphical composition.

The following paragraphs drawn from the book best describe what the Composite pattern does:

*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

*The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphic system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.*

*Gamma/Helm/Johnson/Vlissides, DESIGN PATTERNS, ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, (c) 1995 by Addison-Wesley Publishing Company, Reprinted by permission of Addison-Wesley Publishing Company, Inc.*

Instructions:

Run gdemo as in:

$ wish gdemo

Several buttons are placed below a canvas area. Picture, Rectangle and Oval are used to create Graphic objects. Clear is used to delete all the objects created so far, Exit is self explanatory.

A Picture object can contain any number of Graphic objects, such as other Picture objects, Rectangle objects, ...

For each Graphic object, the point used for moving and for the object coordinates is the upper left corner of the object.

First create a Picture object by clicking on the Picture button. Move the red rectangle that appears by drag clicking on any of its edges. Then create a Rectangle object by clicking on the Rectangle button. Drag the Rectangle object in the Picture object, it is then a child of the Picture object.

Move the Picture object to verify that its Rectangle child moves along.

Create another Picture object and place an Oval object within.

Move that Picture object to verify that its Oval child moves along.

Now move the upper left corner of that last Picture within the first Picture area.

Then move that Picture to verify that all the Graphic objects move along.

#### Widget class

A widget usually can take a variable number of option / value pairs as arguments when created and any time later when configured. It is a good application for the variable number of arguments technique.

Sample code (without error checking):

class widget {  
 proc widget {this parent args} {  
 # create Tk widget(s)  
 # set widget options default in an array  
 array set options {-background white -width 10}  
 array set options $args ;# then overwrite with user options  
 eval configure $this [array get options] ;# then configure  
 }  
 virtual proc configure {this args} {  
 foreach {option value} $args {  
 switch -- $option {  
 -background { ;# filter widget specific options here  
 set ($this,background) $value  
 # configure Tk widget(s)  
 }  
 ...  
 }  
 }  
 }  
}  
  
class gizmo {}  
proc gizmo::gizmo {this parent args} widget {$parent $args} {  
 # create more Tk widget(s)  
 # set gizmo options default in an array  
 array set options {-spacetimecoordinates {0 0 0 now}}  
 array set options $args ;# then overwrite with user options  
 eval ownConfigure $this [array get options] ;# then configure  
}  
proc gizmo::ownConfigure {this args} {  
 foreach {option value} $args {  
 switch -- $option { ;# filter gizmo specific options here  
 -spacetimecoordinates {  
 set ($this,location) $value  
 # configure Tk widget(s)  
 }  
 ...  
 }  
 }  
}  
proc gizmo::configure {this args} {  
 eval ownConfigure $this $args ;# configure at gizmo level  
 eval widget::\_configure $this $args ;# configure at widget level  
}  
  
new gizmo . -width 20 -spacetimecoordinates {1p 10ly 2p 24.2y}

In this example, invalid (unknown) options are simply ignored.

#### Member array

A true member array is not possible, as member data is already held in an array.

But there are 2 ways to get around the problem:

* embed the array and its index in member data
* use a namespace array for possibly better performance

For example, when embedding in member data:

class container {  
 proc container {this} {}  
 proc ~container {this} {}  
 proc container::add {this index value} {  
 set ($this,array,$index) $value  
 }  
}

With a namespace array, use a name specific to the object, including the object identifier, and do not forget to delete the array in the destructor, as the following example shows:

class container {  
 proc container {this} {}  
 proc ~container {this} {  
 variable ${this}array  
 unset ${this}array  
 }  
 proc container::add {this index value} {  
 variable ${this}array  
 set ${this}array($index) $value  
 }  
}

Memory management of the array is the programmer's responsibility, as is its duplication when copying objects. For example, use the following code if you ever copy objects with member arrays:

class container {  
 proc container {this} { ;# main constructor  
 ...  
 } ;# default copy constructor has been generated at this point  
 proc container {this copy} { ;# copy constructor (replaces default one)  
 variable ${this}array  
 variable ${copy}array  
 array set ${this}array [array get ${copy}array] ;# copy member array  
 }  
 ...  
}

### Utility classes

#### switched

***Note****: if you have been using scwoop (a stooop based mega widget extension to the Tk widget library), you must certainly know about the composite class. The switched class is a generic (not widget oriented) derivative of the composite class.*

Find the complete documentation [here](http://docs.google.com/switched.html).

### Debugging

As stooop is meant to be lean and fast, no checking is done during run-time, that is after all classes and their procedures have been defined.

Starting from version 3.3, debugging aids were added to the stooop library (still held in a single file). Member checking insures that basic object oriented concepts and rules are applied. Tracing provides means for member procedures and data access logging to a file or to the screen.

The above features are triggered and configured using environment variables. When not in use, they have absolutely no impact on stooop's performance (however, if you are really picky, you could say that since the stooop.tcl file has grown larger, load time got longer :).

Please note that any stooop debugging environment variable must be set **prior** to the stooop library being loaded:

$ STOOOPTRACEDATA=stdout  
$ export STOOOPTRACEDATA  
$ tclsh myfile.tcl

around the beginning of myfile.tcl:

...  
set env(STOOOPCHECKPROCEDURES) 1  
source stooop.tcl  
namespace import stooop::\*  
set env(STOOOPCHECKDATA) 1  
...

In the example above, data tracing is enabled as well as procedure checking, but data checking is not turned on.

#### Member check

Both procedure and data member checking can be activated by setting the single environment variable STOOOPCHECKALL to a true value (*1*, *true* or *on*). Of course only one of those features can be activated as described below.

*Note: if you have an idea about any other thing that could be checked in the following sections, please share it with* [*me*](mailto:jfontain@free.fr)*.*

##### Procedure

Procedure checking is activated by setting the environment variable STOOOPCHECKPROCEDURES to a true value. The stooop library will then generate an error while the application is running in the following cases:

* an invalid *this* parameter (a non existing object identifier) is passed as argument to a non static member procedure
* the object identified by the *this* parameter passed as argument to a class non static member procedure is neither an instance of the procedure class nor an instance of a derived class (at any level of derivation) of the procedure class.
* a pure interface class (a class with at least 1 pure virtual member procedure) is instanciated

##### Data

Procedure checking is activated by setting the environment variable STOOOPCHECKDATA to a true value. The stooop library will then generate an error while the application is running in the following cases:

* in a class namespace but outside a member procedure, a data member of another class is written or unset
* in a class member procedure (static or not), a data member of another class is written or unset
* in a non static member procedure, a data member of an object different from the object identified by the *this* parameter passed as argument is written or unset

#### Member trace

Tracing is activated by setting a specific environment variable to either *stdout*, *stderr* or any file name that can be created and written to by the user. Setting the STOOOPTRACEALL variable enables both procedure and data tracing. Of course only one of those features can be activated as described below.

##### Procedure

Procedure tracing is activated by setting the environment variable STOOOPTRACEPROCEDURES to either *stdout*, *stderr* or a file name. The stooop library will then output to the specified channel 1 line of informational text for each member procedure invocation.

The user can define the output format by redefining the STOOOPTRACEPROCEDURESFORMAT (look at the beginning of the stooop.tcl file for the default format). The following substitutions will be performed prior to the output:

* **%C** by the fully qualified class name
* **%c** by the class name (tail of the fully qualified class name)
* **%P** by the fully qualified procedure name
* **%p** by the procedure name (tail of the fully qualified procedure name)
* **%O** by the object identifier (*this* value)
* **%a** by the remaining procedure arguments (not including *this*)

At the time this document is being written, the default format is:

class: %C, procedure: %p, object: %O, arguments: %a

example output from the gdemo application:

class: picture, procedure: constructor, object: 1, arguments: .canvas  
class: graphic, procedure: constructor, object: 1, arguments: .canvas 1  
class: rectangle, procedure: constructor, object: 2, arguments: .canvas  
class: graphic, procedure: constructor, object: 2, arguments: .canvas 2  
class: graphic, procedure: moveTo, object: 2, arguments: 13 4  
class: graphic, procedure: \_moveTo, object: 2, arguments: 13 4  
class: graphic, procedure: moveTo, object: 2, arguments: 18 9  
class: graphic, procedure: add, object: 1, arguments: 2  
class: picture, procedure: add, object: 1, arguments: 2  
class: graphic, procedure: add, object: 2, arguments: 2  
class: rectangle, procedure: add, object: 2, arguments: 2  
class: picture, procedure: destructor, object: 1, arguments:  
class: graphic, procedure: destructor, object: 1, arguments:  
class: rectangle, procedure: destructor, object: 2, arguments:  
class: graphic, procedure: destructor, object: 2, arguments:

##### Data

Data tracing is activated by setting the environment variable STOOOPTRACEDATA to either *stdout*, *stderr* or a file name. The stooop library will then output to the specified channel 1 line of informational text for each member data access. By default, all read, write and unsetting accesses are reported, but the user can set the STOOOPTRACEDATAOPERATIONS environment variable to any combination of the *r*, *w* and *u* letters for more specific tracing (please refer to the *trace* Tcl manual page for more information).

Note that operations internal to the stooop library, such as automatic unsetting of data members during objects destruction do not appear in the trace.

The user can define the output format by redefining the STOOOPTRACEDATAFORMAT (look at the beginning of the stooop.tcl file for the default format). The following substitutions will be performed prior to the output:

* **%C** by the fully qualified class name
* **%c** by the class name (tail of the fully qualified class name)
* **%P** by the fully qualified procedure name
* **%p** by the procedure name (tail of the fully qualified procedure name)
* **%A** by the fully qualified array name
* **%m** by the data member name (right after the *this,* array name part for a non static data member)
* **%O** by the object identifier (*this* value or empty for a static procedure)
* **%o** by the access operation (*read*, *write* or *unset*)
* **%v** by the new or current value (empty for an *unset* operation)

At the time this document is being written, the default format is:

class: %C, procedure: %p, array: %A, object: %O, member: %m, operation: %o, value: %v

example output from the gdemo application:

class: graphic, procedure: constructor, array: graphic::, object: 1, member: canvas, operation: write, value: .canvas  
class: graphic, procedure: constructor, array: graphic::, object: 1, member: item, operation: write, value: 1  
class: picture, procedure: constructor, array: picture::, object: 1, member: graphics, operation: write, value:   
class: picture, procedure: moveTo, array: graphic::, object: 1, member: canvas, operation: read, value: .canvas  
class: picture, procedure: moveTo, array: graphic::, object: 1, member: item, operation: read, value: 1  
class: picture, procedure: moveBy, array: picture::, object: 1, member: graphics, operation: read, value:   
class: graphic, procedure: \_moveBy, array: graphic::, object: 1, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: \_moveBy, array: graphic::, object: 1, member: item, operation: read, value: 1  
class: graphic, procedure: constructor, array: graphic::, object: 2, member: canvas, operation: write, value: .canvas  
class: graphic, procedure: constructor, array: graphic::, object: 2, member: item, operation: write, value: 2  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: item, operation: read, value: 2  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: item, operation: read, value: 2  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: item, operation: read, value: 2  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: \_moveTo, array: graphic::, object: 2, member: item, operation: read, value: 2  
class: picture, procedure: add, array: graphic::, object: 1, member: canvas, operation: read, value: .canvas  
class: picture, procedure: add, array: graphic::, object: 1, member: item, operation: read, value: 2  
class: picture, procedure: add, array: graphic::, object: 1, member: canvas, operation: read, value: .canvas  
class: picture, procedure: add, array: graphic::, object: 1, member: item, operation: read, value: 1  
class: graphic, procedure: destructor, array: graphic::, object: 1, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: destructor, array: graphic::, object: 1, member: item, operation: read, value: 1  
class: graphic, procedure: destructor, array: graphic::, object: 2, member: canvas, operation: read, value: .canvas  
class: graphic, procedure: destructor, array: graphic::, object: 2, member: item, operation: read, value: 2

#### Objects

Objects checking can be activated by setting the single environment variable STOOOPCHECKOBJECTS to a true value. The following stooop namespace procedures then become available for debugging; *printObjects*, *record* and *report*.

Before outputting any data, all the object checking procedures print which procedure they were invoked from, or the namespace name if invoked from a namespace body or "*top level*" if invoked outside any procedure or namespace.

##### Printing

The *stooop::printObjects* procedure when invoked prints an ordered list of existing objects with their creation location (a fully qualified procedure name or "*top level*") after a **+** sign. The objects are printed in creation order, with the oldest (lowest identifier) first. The printObjects procedure takes an optional class pattern (as in the Tcl "*array names*" or "*string match*" commands) for limiting the output to objects of certain classes, as the following example shows (classes are assumed to exist and be valid):

% new foo  
1  
% stooop::printObjects  
stooop::printObjects invoked from top level:  
::foo(1) + top level  
% new bar  
2  
% stooop::printObjects  
stooop::printObjects invoked from top level:  
::foo(1) + top level  
::bar(2) + top level  
% new Foo  
3  
% stooop::printObjects ::?oo  
stooop::printObjects invoked from top level:  
::foo(1) + top level  
::Foo(3) + top level  
% new barmaid  
4  
% stooop::printObjects ::bar\*  
stooop::printObjects invoked from top level:  
::bar(2) + top level  
::barmaid(4) + top level

Please note that all object classes are always fully qualified, so do not forget about the **::** header in the patterns.

##### Recording

By invoking the *stooop::record* procedure, you take a snapshot of all existing stooop objects at the time of invocation. Reporting can then be used at a later time to see which objects were created or deleted in the interval.

The record procedure does not take any arguments and it only prints its context of invocation.

##### Reporting

The *stooop::report* procedure prints the created and deleted objects since the stooop::record procedure was invoked last. It optionally takes a pattern argument in order to limit the output to a specific set of classes, as for the printObjects procedure. A **+** sign is placed at the beginning of each created object description line in the output trace, followed by another **+** sign and the creation location (a fully qualified procedure name or "*top level*"). A **-** sign is placed at the beginning of each deleted object description line in the output trace, followed by another **-** sign, the deletion location (a fully qualified procedure name or "*top level*"), a **+** sign and the creation location (a fully qualified procedure name or "*top level*").

Reporting is typically used between 2 spots in the debugged application code: the first spot where a bunch of objects (which can include sub objects) are created, the second spot where all or most of these objects are supposed to be deleted. On the first spot, stooop::record is invoked whereas on the second spot, the stooop::report invocation will print the created and/or deleted objects, in other words the "object difference" between the 2 spots. In most cases, the programmer would expect a difference of 0 objects, sign of a well behaved application, memory wise.

Consider the following example:

class foo {  
 proc foo {this} {}  
 proc ~foo {this} {}  
}  
class bar {  
 proc bar {this} {  
 new foo  
 }  
 proc ~bar {this} {}  
}  
stooop::record  
delete [new bar]  
stooop::report  
stooop::record  
delete 2  
stooop::report

It gives the following result:

stooop::record invoked from top level  
stooop::report invoked from top level:  
+ ::foo(2) + ::bar::bar  
stooop::record invoked from top level  
stooop::report invoked from top level:  
- ::foo(2) - top level + ::bar::bar

Examining the printout, one can see that the bar class does not properly clean things up as the foo sub object is left undeleted.

### Notes

#### On design choices

Performance would have to as good as possible.

A familiar C++ syntax should serve as a model (not all, though, I didn't feel like writing 700 pages of documentation :-).

Tcl being a non declarative language (which I really enjoy), stooop would have to try to comply with that approach.

Error checking would have to be strong with little impact on performance.

#### On implementation

For a Tcl only extension, I think performance is the main issue. The performance / functionality compromise was handled by moving as much processing as possible to the preprocessing stage, handled by the proc and virtual commands. Furthermore, all the costly error checking could be done there as well, having no impact on runtime performance.

The delete operation was greatly simplified, especially for classes that would require a virtual destructor in C++, by storing in an array the class of each object. It then became trivial to delete any object from its identifier only. This approach has an impact on memory use, though, but I consider that one is not very likely to create a huge number of objects in a Tcl application. Furthermore, a classof RTTI operator was then added with no effort.

Stooop learns class hierarchies through the constructor definition whichserves as an implementation as well, thus (kind of) better fitting the non declarative nature of Tcl.

All member data is public but access control is somewhat enforced by having to explicitly name the class layer of external data being accessed.

Since, for performance reasons, the stooop library performs very little checking during run-time (after all classes and their procedures were defined), debugging aids are provided starting from version 3.3. They attempt to insure that your code is well written in an object oriented sense. They also provide means for tracing data access and procedures.

### Miscellaneous information

For downloading other Tcl software (such as scwoop, moodss, ...), visit my [web page](http://jfontain.free.fr/).

Send your comments, complaints, ... to [Jean-Luc Fontaine](mailto:jfontain@free.fr).