NAME

oo::object — root class of the class hierarchy

SYNOPSIS

package require TclOO

oo::object *method* ?*arg* ...?

CLASS HIERARCHY

oo::object

DESCRIPTION

The **oo::object** class is the root class of the object hierarchy; every object is an instance of this class. Since classes are themselves objects, they are instances of this class too. Objects are always referred to by their name, and may be **rename**d while maintaining their identity.

Instances of objects may be made with either the **create** or **new** methods of the **oo::object** object itself, or by invoking those methods on any of the subclass objects; see **oo::class** for more details. The configuration of individual objects (i.e., instance-specific methods, mixed-in classes, etc.) may be controlled with the **oo::objdefine** command.

Each object has a unique namespace associated with it, the instance namespace. This namespace holds all the instance variables of the object, and will be the current namespace whenever a method of the object is invoked (including a method of the class of the object). When the object is destroyed, its instance namespace is deleted. The instance namespace contains the object's **my** command, which may be used to invoke non-exported methods of the object or to create a reference to the object for the purpose of invocation which persists across renamings of the object.

CONSTRUCTOR

The **oo::object** class does not define an explicit constructor.

DESTRUCTOR

The **oo::object** class does not define an explicit destructor.

EXPORTED METHODS

The **oo::object** class supports the following exported methods:

obj **destroy**

This method destroys the object, *obj*, that it is invoked upon, invoking any destructors on the object's class in the process. It is equivalent to using **rename** to delete the object command. The result of this method is always the empty string.

NON-EXPORTED METHODS

The **oo::object** class supports the following non-exported methods:

obj **eval** ?*arg* ...?

This method concatenates the arguments, *arg*, as if with **concat**, and then evaluates the resulting script in the namespace that is uniquely associated with *obj*, returning the result of the evaluation.

Note that object-internal commands such as **my** and **self** can be invoked in this context.

obj **unknown ?***methodName*? ?*arg* ...?
This method is called when an attempt to invoke the method *methodName* on object *obj* fails. The arguments that the user supplied to the method are given as *arg* arguments. If *methodName* is absent, the object was invoked with no method name at all (or any other arguments). The default implementation (i.e., the one defined by the **oo::object** class) generates a suitable error, detailing what methods the object supports given whether the object was invoked by its public name or through the **my** command.

obj **variable** ?*varName* ...?
This method arranges for each variable called *varName* to be linked from the object *obj*'s unique namespace into the caller's context. Thus, if it is invoked from inside a procedure then the namespace variable in the object is linked to the local variable in the procedure. Each *varName* argument must not have any namespace separators in it. The result is the empty string.

obj **varname** *varName*
This method returns the globally qualified name of the variable *varName* in the unique namespace for the object *obj*.

obj **<cloned>** *sourceObjectName*
This method is used by the **oo::object** command to copy the state of one object to another. It is responsible for copying the procedures and variables of the namespace of the source object (*sourceObjectName*) to the current object. It does not copy any other types of commands or any traces on the variables; that can be added if desired by overriding this method in a subclass.

EXAMPLES

This example demonstrates basic use of an object.

```
set obj [oo::object new]
$obj foo             → error "unknown method foo"
oo::objdefine $obj method foo {} {
    my variable count
    puts "bar[incr count]"
}
$obj foo             → prints "bar1"
$obj foo             → prints "bar2"
$obj variable count  → error "unknown method variable"
$obj destroy
$obj foo             → error "unknown command obj"
```

NAME

oo::class — class of all classes

SYNOPSIS

package require TclOO

oo::class *method* ?*arg* ...?

CLASS HIERARCHY

oo::object
→ **oo::class**

DESCRIPTION

Classes are objects that can manufacture other objects according to a pattern stored in the factory object (the class). An instance of the class is created by calling one of the class's factory methods, typically either **create** if an explicit name is being given, or **new** if an arbitrary unique name is to be automatically selected.

The **oo::class** class is the class of all classes; every class is an instance of this class, which is consequently an instance of itself. This class is a subclass of **oo::object**, so every class is also an object. Additional metaclasses (i.e., classes of classes) can be defined if necessary by subclassing **oo::class**. Note that the **oo::class** object hides the **new** method on itself, so new classes should always be made using the **create** method.

CONSTRUCTOR

The constructor of the **oo::class** class takes an optional argument which, if present, is sent to the **oo::define** command (along with the name of the newly-created class) to allow the class to be conveniently configured at creation time.

DESTRUCTOR

The **oo::class** class does not define an explicit destructor. However, when a class is destroyed, all its subclasses and instances are also destroyed, along with all objects that it has been mixed into.

EXPORTED METHODS

cls **create** *name* ?*arg* ...?

This creates a new instance of the class *cls* called *name* (which is resolved within the calling context's namespace if not fully qualified), passing the arguments, *arg* ..., to the constructor, and (if that returns a successful result) returning the fully qualified name of the created object (the result of the constructor is ignored). If the constructor fails (i.e. returns a non-OK result) then the object is destroyed and the error message is the result of this method call.

cls **new** ?*arg* ...?

This creates a new instance of the class *cls* with a new unique name, passing the arguments, *arg* ..., to the constructor, and (if that returns a successful result) returning the fully qualified name of the created object (the result of the constructor is ignored). If the constructor fails (i.e., returns a non-OK result) then the object is destroyed and the error message is the result of this method call.

Note that this method is not exported by the **oo::class** object itself, so classes should not be created using this method.

NON-EXPORTED METHODS

The **oo::class** class supports the following non-exported methods:

cls **createWithNamespace** *name nsName* ?*arg* ...?
: This creates a new instance of the class *cls* called *name* (which is resolved within the calling context's namespace if not fully qualified), passing the arguments, *arg* ..., to the constructor, and (if that returns a successful result) returning the fully qualified name of the created object (the result of the constructor is ignored). The name of the instance's internal namespace will be *nsName*; it is an error if that namespace cannot be created. If the constructor fails (i.e., returns a non-OK result) then the object is destroyed and the error message is the result of this method call.

EXAMPLES

This example defines a simple class hierarchy and creates a new instance of it. It then invokes a method of the object before destroying the hierarchy and showing that the destruction is transitive.

```
oo::class create fruit {
    method eat {} {
        puts "yummy!"
    }
}
oo::class create banana {
    superclass fruit
    constructor {} {
        my variable peeled
        set peeled 0
    }
    method peel {} {
        my variable peeled
        set peeled 1
        puts "skin now off"
    }
    method edible? {} {
        my variable peeled
        return $peeled
    }
    method eat {} {
        if {![my edible?]} {
            my peel
        }
        next
    }
}
set b [banana new]
$b eat               → prints "skin now off" and "yummy!"
fruit destroy
$b eat               → error "unknown command"
```

NAME

oo::define, oo::objdefine, oo::Slot — define and configure classes and objects

SYNOPSIS

package require TclOO

oo::define *class defScript*
oo::define *class subcommand arg* ?*arg* ...?
oo::objdefine *object defScript*
oo::objdefine *object subcommand arg* ?*arg* ...?

oo::Slot *arg...*

CLASS HIERARCHY

oo::object
→ **oo::Slot**

DESCRIPTION

The **oo::define** command is used to control the configuration of classes, and the **oo::objdefine** command is used to control the configuration of objects (including classes as instance objects), with the configuration being applied to the entity named in the *class* or the *object* argument. Configuring a class also updates the configuration of all subclasses of the class and all objects that are instances of that class or which mix it in (as modified by any per-instance configuration). The way in which the configuration is done is controlled by either the *defScript* argument or by the *subcommand* and following *arg* arguments; when the second is present, it is exactly as if all the arguments from *subcommand* onwards are made into a list and that list is used as the *defScript* argument.

CONFIGURING CLASSES

The following commands are supported in the *defScript* for **oo::define**, each of which may also be used in the *subcommand* form:

constructor *argList bodyScript*
: This creates or updates the constructor for a class. The formal arguments to the constructor (defined using the same format as for the Tcl **proc** command) will be *argList*, and the body of the constructor will be *bodyScript*. When the body of the constructor is evaluated, the current namespace of the constructor will be a namespace that is unique to the object being constructed. Within the constructor, the **next** command should be used to call the superclasses' constructors. If *bodyScript* is the empty string, the constructor will be deleted.

deletemethod *name* ?*name* ...?
: This deletes each of the methods called *name* from a class. The methods must have previously existed in that class. Does not affect the superclasses of the class, nor does it affect the subclasses or instances of the class (except when they have a call chain through the class being modified) or the class object itself.

destructor *bodyScript*

This creates or updates the destructor for a class. Destructors take no arguments, and the body of the destructor will be *bodyScript*. The destructor is called when objects of the class are deleted, and when called will have the object's unique namespace as the current namespace. Destructors should use the **next** command to call the superclasses' destructors. Note that destructors are not called in all situations (e.g. if the interpreter is destroyed). If *bodyScript* is the empty string, the destructor will be deleted.

Note that errors during the evaluation of a destructor *are not returned* to the code that causes the destruction of an object. Instead, they are passed to the currently-defined **bgerror** handler.

export *name* ?*name* ...?

This arranges for each of the named methods, *name*, to be exported (i.e. usable outside an instance through the instance object's command) by the class being defined. Note that the methods themselves may be actually defined by a superclass; subclass exports override superclass visibility, and may in turn be overridden by instances.

filter ?-*slotOperation*? ?*methodName* ...?

This slot (see **SLOTTED DEFINITIONS** below) sets or updates the list of method names that are used to guard whether method call to instances of the class may be called and what the method's results are. Each *methodName* names a single filtering method (which may be exposed or not exposed); it is not an error for a non-existent method to be named since they may be defined by subclasses. By default, this slot works by appending.

forward *name cmdName* ?*arg* ...?

This creates or updates a forwarded method called *name*. The method is defined be forwarded to the command called *cmdName*, with additional arguments, *arg* etc., added before those arguments specified by the caller of the method. The *cmdName* will always be resolved using the rules of the invoking objects' namespaces, i.e., when *cmdName* is not fully-qualified, the command will be searched for in each object's namespace, using the instances' namespace's path, or by looking in the global namespace. The method will be exported if *name* starts with a lower-case letter, and non-exported otherwise.

method *name argList bodyScript*

This creates or updates a method that is implemented as a procedure-like script. The name of the method is *name*, the formal arguments to the method (defined using the same format as for the Tcl **proc** command) will be *argList*, and the body of the method will be *bodyScript*. When the body of the method is evaluated, the current namespace of the method will be a namespace that is unique to the current object. The method will be exported if *name* starts with a lower-case letter, and non-exported otherwise; this behavior can be overridden via **export** and **unexport**.

mixin ?-*slotOperation*? ?*className* ...?

This slot (see **SLOTTED DEFINITIONS** below) sets or updates the list of additional classes that are to be mixed into all the instances of the class being defined. Each *className* argument names a single class that is to be mixed in. By default, this slot works by replacement.

renamemethod *fromName toName*

This renames the method called *fromName* in a class to *toName*. The method must have previously existed in the class, and *toName* must not previously refer to a method in that class. Does not affect the superclasses of the class, nor does it affect the subclasses or instances of the class (except when they have a call chain through the class being modified), or the class object itself. Does not change the export status of the method; if it was exported before, it will be afterwards.

self *subcommand arg ...*

self *script*

This command is equivalent to calling **oo::objdefine** on the class being defined (see **CONFIGURING OBJECTS** below for a description of the supported values of *subcommand*). It follows the same general pattern of argument handling as the **oo::define** and **oo::objdefine** commands, and "**oo::define** *cls* **self** *subcommand ...*" operates identically to "**oo::objdefine** *cls subcommand ...*".

superclass ?-*slotOperation*? ?*className ...*?

This slot (see **SLOTTED DEFINITIONS** below) allows the alteration of the superclasses of the class being defined. Each *className* argument names one class that is to be a superclass of the defined class. Note that objects must not be changed from being classes to being non-classes or vice-versa, that an empty parent class is equivalent to **oo::object**, and that the parent classes of **oo::object** and **oo::class** may not be modified. By default, this slot works by replacement.

unexport *name* ?*name ...*?

This arranges for each of the named methods, *name*, to be not exported (i.e. not usable outside the instance through the instance object's command, but instead just through the **my** command visible in each object's context) by the class being defined. Note that the methods themselves may be actually defined by a superclass; subclass unexports override superclass visibility, and may be overridden by instance unexports.

variable ?-*slotOperation*? ?*name ...*?

This slot (see **SLOTTED DEFINITIONS** below) arranges for each of the named variables to be automatically made available in the methods, constructor and destructor declared by the class being defined. Each variable name must not have any namespace separators and must not look like an array access. All variables will be actually present in the instance object on which the method is executed. Note that the variable lists declared by a superclass or subclass are completely disjoint, as are variable lists declared by instances; the list of variable names is just for methods (and constructors and destructors) declared by this class. By default, this slot works by appending.

CONFIGURING OBJECTS

The following commands are supported in the *defScript* for **oo::objdefine**, each of which may also be used in the *subcommand* form:

class *className*

This allows the class of an object to be changed after creation. Note that the class's constructors are not called when this is done, and so the object may well be in an inconsistent state unless additional configuration work is done.

deletemethod *name* ?*name ...*

This deletes each of the methods called *name* from an object. The methods must have previously existed in that object (e.g., because it was created through **oo::objdefine method**). Does not affect the classes that the object is an instance of, or remove the exposure of those class-provided methods in the instance of that class.

export *name* ?*name* ...?

This arranges for each of the named methods, *name*, to be exported (i.e. usable outside the object through the object's command) by the object being defined. Note that the methods themselves may be actually defined by a class or superclass; object exports override class visibility.

filter ?-*slotOperation*? ?*methodName* ...?

This slot (see **SLOTTED DEFINITIONS** below) sets or updates the list of method names that are used to guard whether a method call to the object may be called and what the method's results are. Each *methodName* names a single filtering method (which may be exposed or not exposed); it is not an error for a non-existent method to be named. Note that the actual list of filters also depends on the filters set upon any classes that the object is an instance of. By default, this slot works by appending.

forward *name cmdName* ?*arg* ...?

This creates or updates a forwarded object method called *name*. The method is defined be forwarded to the command called *cmdName*, with additional arguments, *arg* etc., added before those arguments specified by the caller of the method. Forwarded methods should be deleted using the **method** subcommand. The method will be exported if *name* starts with a lower-case letter, and non-exported otherwise.

method *name argList bodyScript*

This creates, updates or deletes an object method. The name of the method is *name*, the formal arguments to the method (defined using the same format as for the Tcl **proc** command) will be *argList*, and the body of the method will be *bodyScript*. When the body of the method is evaluated, the current namespace of the method will be a namespace that is unique to the object. The method will be exported if *name* starts with a lower-case letter, and non-exported otherwise.

mixin ?-*slotOperation*? ?*className* ...?

This slot (see **SLOTTED DEFINITIONS** below) sets or updates a per-object list of additional classes that are to be mixed into the object. Each argument, *className*, names a single class that is to be mixed in. By default, this slot works by replacement.

renamemethod *fromName toName*

This renames the method called *fromName* in an object to *toName*. The method must have previously existed in the object, and *toName* must not previously refer to a method in that object. Does not affect the classes that the object is an instance of and cannot rename in an instance object the methods provided by those classes (though a **oo::objdefine forward**ed method may provide an equivalent capability). Does not change the export status of the method; if it was exported before, it will be afterwards.

unexport *name* ?*name* ...?

This arranges for each of the named methods, *name*, to be not exported (i.e. not usable outside the object through the object's command, but instead just through the **my** command visible in the object's context) by the object being defined. Note that the methods themselves may be actually defined by a class; instance unexports override class visibility.

variable ?-*slotOperation*? ?*name* ...?

This slot (see **SLOTTED DEFINITIONS** below) arranges for each of the named variables to be automatically made available in the methods declared by the object being defined. Each variable name must not have any namespace separators and must not look like an array access. All variables will be actually present in the object on which the method is executed. Note that the variable lists declared by the classes and mixins of which the object is an instance are completely disjoint; the list of variable names is just for methods declared by this object. By default, this slot works by appending.

SLOTTED DEFINITIONS

Some of the configurable definitions of a class or object are *slotted definitions*. This means that the configuration is implemented by a slot object, that is an instance of the class **oo::Slot**, which manages a list of values (class names, variable names, etc.) that comprises the contents of the slot.

The **oo::Slot** class defines three operations (as methods) that may be done on the slot:

slot **-append** ?*member* ...?
: This appends the given *member* elements to the slot definition.

slot **-clear**
: This sets the slot definition to the empty list.

slot **-set** ?*member* ...?
: This replaces the slot definition with the given *member* elements.

A consequence of this is that any use of a slot's default operation where the first member argument begins with a hyphen will be an error. One of the above operations should be used explicitly in those circumstances.

You only need to make an instance of **oo::Slot** if you are definining your own slot that behaves like a standard slot.

SLOT IMPLEMENTATION

Internally, slot objects also define a method **--default-operation** which is forwarded to the default operation of the slot (thus, for the class "**variable**" slot, this is forwarded to "**my -append**"), and these methods which provide the implementation interface:

slot **Get**
: Returns a list that is the current contents of the slot. This method must always be called from a stack frame created by a call to **oo::define** or **oo::objdefine**.

slot **Resolve** *element*
: This converts an element of the slotted collection into its resolved form; for a simple value, it could just return the value, but for a slot that contains references to commands or classes it should convert those into their fully-qualified forms (so they can be compared with **string equals**): that could be done by forwarding to **namespace which** or similar.

slot **Set** *elementList*
: Sets the contents of the slot to the list *elementList* and returns the empty string. This method must always be called from a stack frame created by a call to **oo::define** or **oo::objdefine**.

The implementation of these methods is slot-dependent (and responsible for accessing the correct part of the class or object definition). Slots also have an unknown method handler to tie all these pieces together, and they hide their **destroy** method so that it is not invoked inadvertently. It is *recommended* that any user changes to the slot mechanism itself be restricted to defining new operations whose names start with a hyphen.

Note that slot instances are not expected to contain the storage for the slot they manage; that will be in or attached to the class or object that they manage. Those instances should provide their own implementations of the **Get** and **Set** methods (and optionally **Resolve**; that defaults to a do-nothing pass-through).

EXAMPLES

This example demonstrates how to use both forms of the **oo::define** and **oo::objdefine** commands (they work in the same way), as well as illustrating four of the subcommands of them.

```
oo::class create c
c create o
oo::define c method foo {} {
    puts "world"
}
oo::objdefine o {
    method bar {} {
        my Foo "hello "
        my foo
    }
    forward Foo ::puts -nonewline
    unexport foo
}
o bar                 → prints "hello world"
o foo                 → error "unknown method foo"
o Foo Bar             → error "unknown method Foo"
oo::objdefine o renamemethod bar lollipop
o lollipop            → prints "hello world"
```

This example shows how additional classes can be mixed into an object. It also shows how **mixin** is a slot that supports appending:

```
oo::object create inst
inst m1               → error "unknown method m1"
inst m2               → error "unknown method m2"

oo::class create A {
    method m1 {} {
        puts "red brick"
    }
}
oo::objdefine inst {
    mixin A
}
inst m1               → prints "red brick"
inst m2               → error "unknown method m2"

oo::class create B {
    method m2 {} {
        puts "blue brick"
    }
}
oo::objdefine inst {
    mixin -append B
}
inst m1               → prints "red brick"
inst m2               → prints "blue brick"
```

NAME

my — invoke any method of current object

SYNOPSIS

package require TclOO

my *methodName* ?*arg* ...?

DESCRIPTION

The **my** command is used to allow methods of objects to invoke any method of the object (or its class). In particular, the set of valid values for *methodName* is the set of all methods supported by an object and its superclasses, including those that are not exported. The object upon which the method is invoked is always the one that is the current context of the method (i.e. the object that is returned by **self object**) from which the **my** command is invoked.

Each object has its own **my** command, contained in its instance namespace.

EXAMPLES

This example shows basic use of **my** to use the **variables** method of the **oo::object** class, which is not publicly visible by default:

```
oo::class create c {
    method count {} {
        my variable counter
        puts [incr counter]
    }
}
c create o
o count                  → prints "1"
o count                  → prints "2"
o count                  → prints "3"
```

NAME

next, nextto — invoke superclass method implementations

SYNOPSIS

package require TclOO

next ?*arg ...*?
nextto *class* ?*arg ...*?

DESCRIPTION

The **next** command is used to call implementations of a method by a class, superclass or mixin that are overridden by the current method. It can only be used from within a method. It is also used within filters to indicate the point where a filter calls the actual implementation (the filter may decide to not go along the chain, and may process the results of going along the chain of methods as it chooses). The result of the **next** command is the result of the next method in the method chain; if there are no further methods in the method chain, the result of **next** will be an error. The arguments, *arg*, to **next** are the arguments to pass to the next method in the chain.

The **nextto** command is the same as the **next** command, except that it takes an additional *class* argument that identifies a class whose implementation of the current method chain (see **info object call**) should be used; the method implementation selected will be the one provided by the given class, and it must refer to an existing non-filter invocation that lies further along the chain than the current implementation.

THE METHOD CHAIN

When a method of an object is invoked, things happen in several stages:

1. The structure of the object, its class, superclasses, filters, and mixins, are examined to build a *method chain*, which contains a list of method implementations to invoke.
2. The first method implementation on the chain is invoked.
3. If that method implementation invokes the **next** command, the next method implementation is invoked (with its arguments being those that were passed to **next**).
4. The result from the overall method call is the result from the outermost method implementation; inner method implementations return their results through **next**.
5. The method chain is cached for future use.

METHOD SEARCH ORDER

When constructing the method chain, method implementations are searched for in the following order:

1. In the classes mixed into the object, in class traversal order. The list of mixins is checked in natural order.

2. In the classes mixed into the classes of the object, with sources of mixing in being searched in class traversal order. Within each class, the list of mixins is processed in natural order.
3. In the object itself.
4. In the object's class.
5. In the superclasses of the class, following each superclass in a depth-first fashion in the natural order of the superclass list.

Any particular method implementation always comes as *late* in the resulting list of implementations as possible; this means that if some class, A, is both mixed into a class, B, and is also a superclass of B, the instances of B will always treat A as a superclass from the perspective of inheritance. This is true even when the multiple inheritance is processed indirectly.

FILTERS

When an object has a list of filter names set upon it, or is an instance of a class (or has mixed in a class) that has a list of filter names set upon it, before every invocation of any method the filters are processed. Filter implementations are found in class traversal order, as are the lists of filter names (each of which is traversed in natural list order). Explicitly invoking a method used as a filter will cause that method to be invoked twice, once as a filter and once as a normal method.

Each filter should decide for itself whether to permit the execution to go forward to the proper implementation of the method (which it does by invoking the **next** command as filters are inserted into the front of the method call chain) and is responsible for returning the result of **next**.

Filters are invoked when processing an invocation of the **unknown** method because of a failure to locate a method implementation, but *not* when invoking either constructors or destructors. (Note however that the **destroy** method is a conventional method, and filters are invoked as normal when it is called.)

EXAMPLES

This example demonstrates how to use the **next** command to call the (super)class's implementation of a method. The script:

```
oo::class create theSuperclass {
    method example {args} {
        puts "in the superclass, args = $args"
    }
}
oo::class create theSubclass {
    superclass theSuperclass
    method example {args} {
        puts "before chaining from subclass, args = $args"
        next a {*}$args b
        next pureSynthesis
        puts "after chaining from subclass"
    }
}
theSubclass create obj
oo::objdefine obj method example args {
    puts "per-object method, args = $args"
    next x {*}$args y
    next
}
```

```
obj example 1 2 3
```

prints the following:

```
per-object method, args = 1 2 3
before chaining from subclass, args = x 1 2 3 y
in the superclass, args = a x 1 2 3 y b
in the superclass, args = pureSynthesis
after chaining from subclass
before chaining from subclass, args =
in the superclass, args = a b
in the superclass, args = pureSynthesis
after chaining from subclass
```

This example demonstrates how to build a simple cache class that applies memoization to all the method calls of the objects it is mixed into, and shows how it can make a difference to computation times:

```
oo::class create cache {
    filter Memoize
    method Memoize args {
        # Do not filter the core method implementations
        if {[lindex [self target] 0] eq "::oo::object"} {
            return [next {*}$args]
        }

        # Check if the value is already in the cache
        my variable ValueCache
        set key [self target],$args
        if {[info exist ValueCache($key)]} {
            return $ValueCache($key)
        }

        # Compute value, insert into cache, and return it
        return [set ValueCache($key) [next {*}$args]]
    }
    method flushCache {} {
        my variable ValueCache
        unset ValueCache
        # Skip the caching
        return -level 2 ""
    }
}

oo::object create demo
oo::objdefine demo {
    mixin cache
    method compute {a b c} {
        after 3000 ;# Simulate deep thought
        return [expr {$a + $b * $c}]
    }
    method compute2 {a b c} {
        after 3000 ;# Simulate deep thought
        return [expr {$a * $b + $c}]
    }
}

puts [demo compute  1 2 3]      → prints "7" after delay
puts [demo compute2 4 5 6]      → prints "26" after delay
puts [demo compute  1 2 3]      → prints "7" instantly
puts [demo compute2 4 5 6]      → prints "26" instantly
puts [demo compute  4 5 6]      → prints "34" after delay
puts [demo compute  4 5 6]      → prints "34" instantly
puts [demo compute  1 2 3]      → prints "7" instantly
demo flushCache
puts [demo compute  1 2 3]      → prints "7" after delay
```

NAME

self — method call internal introspection

SYNOPSIS

package require TclOO

self ?*subcommand*?

DESCRIPTION

The **self** command, which should only be used from within the context of a call to a method (i.e. inside a method, constructor or destructor body) is used to allow the method to discover information about how it was called. It takes an argument, *subcommand*, that tells it what sort of information is actually desired; if omitted the result will be the same as if **self object** was invoked. The supported subcommands are:

self call

This returns a two-element list describing the method implementations used to implement the current call chain. The first element is the same as would be reported by **info object** **call** for the current method (except that this also reports useful values from within constructors and destructors, whose names are reported as **<constructor>** and **<destructor>** respectively), and the second element is an index into the first element's list that indicates which actual implementation is currently executing (the first implementation to execute is always at index 0).

self caller

When the method was invoked from inside another object method, this subcommand returns a three element list describing the containing object and method. The first element describes the declaring object or class of the method, the second element is the name of the object on which the containing method was invoked, and the third element is the name of the method (with the strings **<constructor>** and **<destructor>** indicating constructors and destructors respectively).

self class

This returns the name of the class that the current method was defined within. Note that this will change as the chain of method implementations is traversed with **next**, and that if the method was defined on an object then this will fail.

If you want the class of the current object, you need to use this other construct:

```
info object class [self object]
```

self filter

When invoked inside a filter, this subcommand returns a three element list describing the filter. The first element gives the name of the object or class that declared the filter (note that this may be different from the object or class that provided the implementation of the filter), the second element is either **object** or **class** depending on whether the declaring entity was an object or class, and the third element is the name of the filter.

self method

This returns the name of the current method (with the strings **<constructor>** and **<destructor>** indicating constructors and destructors respectively).

self namespace

This returns the name of the unique namespace of the object that the method was invoked upon.

self next

When invoked from a method that is not at the end of a call chain (i.e. where the **next** command will invoke an actual method implementation), this subcommand returns a two element list describing the next element in the method call chain; the first element is the name of the class or object that declares the next part of the call chain, and the second element is the name of the method (with the strings **<constructor>** and **<destructor>** indicating constructors and destructors respectively). If invoked from a method that is at the end of a call chain, this subcommand returns the empty string.

self object

This returns the name of the object that the method was invoked upon.

self target

When invoked inside a filter implementation, this subcommand returns a two element list describing the method being filtered. The first element will be the name of the declarer of the method, and the second element will be the actual name of the method.

EXAMPLES

This example shows basic use of **self** to provide information about the current object:

```
oo::class create c {
    method foo {} {
        puts "this is the [self] object"
    }
}
c create a
c create b
a foo                 → prints "this is the ::a object"
b foo                 → prints "this is the ::b object"
```

This demonstrates what a method call chain looks like, and how traversing along it changes the index into it:

```
oo::class create c {
    method x {} {
        puts "Cls: [self call]"
    }
}
c create a
oo::objdefine a {
    method x {} {
        puts "Obj: [self call]"
        next
        puts "Obj: [self call]"
    }
}
a x       → Obj: {{method x object method} {method x ::c method}} 0
          → Cls: {{method x object method} {method x ::c method}} 1
          → Obj: {{method x object method} {method x ::c method}} 0
```