TIP 257: Object Orientation for Tcl

Abstract

This TIP proposes adding OO support to the Tcl core, semantically inspired by XOTcl. The commands it defines will be in the **::oo** namespace, which is not used by any current mainstream OO system, and it will be designed specifically to allow other object systems to be built on top.

Rationale and Basic Requirements

Tcl has a long history of being comparatively agnostic about object-oriented programming, not favouring one OO system over another while promoting a wealth of OO extensions such as incr Tcl, OTcl, XOTcl, stooop, Snit, etc. because in general, one size fits nobody.

However, many application domains require OO systems and having a common such base system will help prevent application and library authors from reinventing the wheel each time through because they cannot rely on an OO framework being present with each and every Tcl installation. For example, the http package supplied with Tcl has its own internal object model, and a similar mechanism is reinvented multiple times within tcllib. Other parts of tcllib do their own thing (to say nothing of the fact that both stooop and Snit are in tcllib themselves). This does not promote efficient reuse of each others code, and ensures that each of these packages has a *poor* object system. The request for an OO system is also one of the biggest feature requests for Tcl, and would make it far easier to implement megawidgets. It also leaves Tcl open to the ill-informed criticism that it doesn't support OO, despite being spoilt for choice in reality through the extensions listed above.

Given all this, the time has come for the core to provide OO support. The aim of the core OO system shall be that it is simple to get started with, flexible so that it can take you a long way, fast (we all know that we're going to get compared on this front!), and suitable for use as a foundation of many other things, including the re-implementation of various existing OO extensions, including those that are currently compiled and also those that are pure Tcl extensions.

Another requirement is that programmers should not have to alter all of their existing code in order to get started with the new system; rather, they should be able to adopt it progressively, over time, because it supports better ways of working (e.g., faster and more flexible libraries).

The Foundational OO System

This TIP proposes that the foundation of the OO system should ensure that it is simple, fast and flexible. Semantically, the OO system should be using the semantic model pioneered by OTcl and XOTcl, as leveraging their experience on the complex parts (e.g., the model of multiple inheritance, how to invoke superclass implementations of a method) allows us to go straight to a solution that is rich enough for a very large space of applications.

However, some changes relative to XOTcl are necessary. Certain aspects of XOTcl syntax are peculiar from a conventional OO point-of-view, and it is deeply unfortunate that a very large number of methods are predefined in the XOTcl base class. XOTcl's approach to object creation options is also highly idiosyncratic (though critical to the way XOTcl itself works) and doesn't really support the typical Tcl idioms. The changes must be made in such a way that something that works like classic XOTcl for virtually all uses can be built on the new framework, but the core object framework must also enable building [incr Tcl]-like or Snit-like object systems on top.

Note that by keeping things in the base classes comparatively simple, it is much easier to build multiple extended OO frameworks on top.

Key Features

Functional Requirements

- Class-based object system. This is what most programmers expect from OO, and it is very useful for many tasks.
- Allows per-object customization and dynamic redefinition of classes.
- Supports advanced OO features, such as:

  meta-classes: These are subclasses of **class**, which permit more advanced customization of class behaviour.

  filters: These are constraints (implemented in Tcl code, naturally) on whether a method may be called.

  mixins: These allow functionality to be brought into an object from other objects if necessary, enabling better separation of concerns.

- A system for implementing methods in custom ways, so that package authors that want significantly different ways of doing a method implementation may do so fairly simply. Note that this will require additional C code to perform; this API will not be exposed directly to the script level (since it makes little sense there).

Non-Functional Requirements

Note that these requirements would need to be imposed on any implementation of an object system in the Tcl core code anyway.

- The speed of the object system is something on which it is easy to predict that Tcl will end up being compared to other languages. Hence, the core OO system *must* permit efficient implementation.
- The core OO system must be clear code that it is easy for the Tcl maintainers to keep in good order. The Engineering Manual will be followed.

Key Alterations Relative to XOTcl

The core OO system can be considered to be a derivative of XOTcl, much as C can be considered to be a derivative of Algol. However, like the C/Algol relationship, there are many changes between the core OO system and XOTcl; the *implementations* are not common.

- Object and class names in the core extension to be all lower-case, in line with best common practice in general Tcl code.
- Methods have to be capable of being non-exported, by which we mean that they are not (simply) callable from contexts outside the object.
- The majority of the API for updating an object or class's definition is to be moved to a separate utility command, **oo::define**.
- More "conventional" naming of operations is to be used.
- Many of the more advanced features of XOTcl are not present, especially when it is possible to implement them on top of other features. This particularly applies to:

  * Filter- and mixin-guards

  * Invariants

  * Pre- and post-conditions

Note that this TIP does *not* propose to actually include any XOTcl (or Itcl or Snit or ...) compatibility packages in the core; it is about forming a foundation on which they can be built (which happens to also be a comparatively lightweight OO system in itself). Such compatibility packages can either remain separate code, or be the subject of future TIPs.

Detailed Rationale for Not Using XOTcl

Features of XOTcl that are Retained

Many key semantic features of XOTcl are adopted with little or no change. In particular, the following critical features of the core object system shall be semantically the same as in XOTcl as they represent the best-of-breed in advanced object systems at the moment.

Multiple Inheritance

We shall support multiple inheritance (MI) because this is very difficult to add after the fact.

The main problem with MI in languages like C++ was always confusion caused by the fact that methods were resolved using integer offsets into method tables. By contrast, single inheritance is far too restrictive. By supporting mixins and filters, it becomes possible to build not just conventional OO systems in the C++ or Java mould, but also to make Self-like prototype systems (which requires mixins and subclassing of the class of classes to work) and Aspect-like systems (which require filters for efficient implementation). As these are less well-known terms than those of normal inheritance, we define them here:

mixin: An auxiliary *class* whose behaviour is "mixed into" the current object or class, adding the mixin's methods to the target's methods. Often used to support cross-cutting functionality or object roles.

filter: A nominated *method* that is permitted to control whether all calls to any other method of a class or object occur. This control is achieved by the nominated filter method being chained on the front of the sequence of methods in the "implementation list" for the actual target method. Often used to support transparent orthogonal functionality, such as access control or result caching.

The Method Dispatch Algorithm

The use of a complex class graph model as described above requires a sophisticated algorithm to linearize any particular call of a method into a sequence of method implementations that should be called. This is the method dispatch algorithm. The algorithm works by scanning the graph of the object and its associated classes in the order given below, collecting implementations as they are found in a list, with an implementation coming in the *last* position in the list it can be; if it would be in twice because of an "inheritance diamond", it comes in the later location.

1. Filters defined on classes mixed into the object or its class (or superclasses), with filters from a particular class processed in the order that they are described by that class.
2. Filters defined on the object, in the order that they are described in the object's filter list.
3. Filters defined on the class of the object (or its superclasses), with filters from a particular class processed in the order that they are described by that class.
4. Methods declared by mixins added to the object, with mixins being processed in the order that they are described in the object's mixin list (the set of methods declared by the mixin are determined by recursively applying this algorithm).
5. Methods declared by mixins added to the object's class or superclasses, with mixins to a particular class being processed in the order that they are described in the class's mixin list.
6. Methods declared by the object itself.
7. Methods declared by the object's class itself.
8. Methods declared by the object's class's superclasses, with superclasses being processed in the order that they are described in the class's superclass list.

Given the above ordering, for each method on an object there is an ordered list of implementations. We dispatch the method by executing the first implementation on the list, which can then hand off to subsequent methods in the list in order using the **next** command (described below).

Another way to view the ordering is that there are several layers to the ordering scheme. Firstly, there is the basic ordering which is: object, class, superclasses to root (with multiple inheritance being processed in the listed order, and classes appearing in the linearized tree as late as possible). Then there is the second-order ordering, which adds mixins to the front of the basic ordering, where the order of the mixins is processed in basic ordering. Finally, there is the third-order ordering which adds filters to the front of the second-order ordering, with filter name sources being processed in second-order ordering and the method chain for a particular filter being processed in second-order ordering order.

Note that a filter being invoked as a filter is different to a filter being invoked as a normal method. If a filter method is invoked directly, then it will actually be invoked twice, once as a filter and once as a conventional method.

Essential Changes Relative to XOTcl

Unfortunately, not all features of XOTcl are suitable for a core object system. In particular, many more syntactic features need to be altered. This section describes these, together with the rationale for each change; the rationales are marked with the word "**Therefore**", in bold.

Exported vs. Non-exported Methods

In XOTcl, every class and every object has an associated namespace. The namespace associated with a class *::myclass* is *::xotcl::classes::myclass*; the namespace associated with object *::myobject* is simply *::myobject*. XOTcl "instprocs" are simply procs defined in a class (or superclass) namespace; XOTcl per-object "procs" are simply procs defined in an object's namespace. *Every such proc becomes an object subcommand.*

This is part of the reason why XOTcl objects have such cluttered interfaces. Every method which is of use to the object appears in the object's interface - and there's no way to prevent this.

Therefore, in the new oo system "**proc**s" and "**instproc**s" can be exported or non-exported. Exported procs appear as object subcommands; non-exported procs do not, but remain available as subcommands of the **my** command. In this way, the object itself can still use them, but they need appear in the object's interface only if desired.

Additionally, the standard introspection system will need to be extended to allow determining of which methods are exported and which are not.

The oo::define Command

In XOTcl, the commands to define per-class methods, filters, and so on are subcommands of the class object; the commands to define per-object methods, filters, and so on are subcommands of the individual object. This is a problem, as it confuses the implementation-time interface with the run-time interface. The design is logical, given XOTcl's extreme dynamism; any implementation-time activity, such as defining a method or adding a filter can indeed be done at run-time. But again, this makes it difficult to define clean run-time interfaces for reusable library code.

The solution described in the previous section, of making some methods private by declaring them non-exported, does not give us a full solution; having the **instproc** subcommand available only from instance code isn't all that useful.

Therefore, we add two new commands, **oo::define** and **oo::objdefine**, which are used to define methods, filters, and so on. They can be called in two ways, with each command having essentially the same fundamental syntax. The first calling model is as follows:

> **oo::define** *class subcommand args...*
>
> **oo::objdefine** *object subcommand args...*

For example, the following XOTcl code defines a class with two methods:

```
xotcl::Class myclass
myclass instproc dothis {args} { # body }
myclass instproc dothat {args} { # body }
```

In the new oo core, the matching code would be this:

```
oo::class create myclass
oo::define myclass method dothis {args} { # body }
oo::define myclass method dothat {args} { # body }
```

The two definition commands will also have a second calling model, in which they get passed a single definition script whose commands are the subcommands supported by the command as described above:

> **oo::define** *class script*
>
> **oo::objdefine** *object script*

Thus, the above code could also be written as follows:

```
oo::class create myclass
oo::define myclass {
    method dothis {args} { # body }
    method dothat {args} { # body }
}
```

Finally, the constructor for the **oo::class** class is extended so that such a script can be used during class creation:

```
oo::class create myclass {
    method dothis {args} { # body }
    method dothat {args} { # body }
}
```

This allows a class to be defined cleanly and concisely, while guaranteeing that all class details can still be modified later on using **oo::define**.

To enable the easy definition of details of the class object at class definition time, a special subcommand, "**self**", will be provided that is equivalent to using **oo::objdefine** on the class.

Note that because of the requirement for a distinction between public and private interfaces, **oo::define** and **oo::objdefine** will need two subcommands XOTcl doesn't currently provide: **export** and **unexport**. **export** takes as arguments a list of method names; all named methods are exported and become visible in the object or class's interface. **unexport** does the opposite. Note that by default, all methods that start with a lower-case letter (specifically, names matching the glob pattern "[a-z]*") will be exported by default and all other methods will be unexported.

Standard Metaclasses

XOTcl defines two standard Metaclasses, *xotcl::Object* and *xotcl::Class*. *xotcl::Object* is the root of the class hierarchy; all XOTcl classes implicitly inherit from *xotcl::Object*. XOTcl classes are themselves objects, and are instances of *xotcl::Class*. *xotcl::Class* can itself be subclassed to produce different families of classes with different standard behaviours.

The new core object system will use the same basic mechanism, based on the metaclasses **oo::object** and **oo::class**. However, one of the problems with XOTcl is that XOTcl objects have too much standard behavior; the new core object system must provide a simpler foundation, with the XOTcl behavior optionally available.

Therefore, we will extract the features of *xotcl::Object* and *xotcl::Class* that are critical into our classes and leave all other functionality up to any subclasses or metaclasses that are defined.

Thus **oo::object** will be the root of the class hierarchy. However, instances of **oo::object** will have a minimal set of standard methods, so that clean interfaces can be built on top of it, as can be done with Snit types and instances.

Core object system classes will be instances of **oo::class** or its subclasses. Likewise, **oo::class** will define only minimal behaviour.

Inheritance

A class may wish to make use of the capabilities of **oo::class** internally without exporting its methods (e.g., for providing a singleton instance).

Therefore, the inheritance mechanism should be extended such that the newly defined class can declare whether a parent class's methods should be exported or not, on a case-by-case basis.

Object Creation

XOTcl has a unique creation syntax. The object name can be followed by what look like Tk or Snit options - but aren't. Instead, any token in the argument list that begins with a hyphen is assumed to be the name of one of the object's methods; it must be followed by the method's own arguments. For example, a standard XOTcl class will have a "set" method, which has the same syntax as the standard Tcl "set" command. Thus, the following code:

```
myclass myobj -set a 1 -set b 2
```

creates an instance of "myclass" called "myobj" whose instance variables "a" and "b" and set to 1 and 2 respectively. This is an intriguing and innovative interface, and it is unlike any other Tcl object system. Additionally, it makes it difficult to implement standard Tk-like options.

Therefore, standard core object system classes will not use this mechanism (though it might be available on demand by inheriting from some other standard metaclass). Instead, standard core object system classes will have no creation behavior other than that implemented by their designers in their constructors.

Constructors may have any argument list the user pleases, including default arguments, the "args" argument (as in the **proc** command), and XOTcl-style non-positional arguments. It is up to the developer to handle the arguments appropriately.

It is expected that one of the key responsibilities of any XOTcl compatability package would be to define an object/class construction system that parses the arguments in the expected way and uses them to invoke methods on the newly created object.

Constructor Syntax

In XOTcl, a class's constructor is implemented using its "init" instproc. This is troubling; constructors are intended to do things just once, and are often written to take advantage of that, whereas an "init" instproc can theoretically be called at any time. For any given class, then, one of two conditions will obtain: either "init" must be written so that it can be called at any time, or the class will have an inherent logic bug.

Therefore, the class constructor will not be implemented as a standard instproc. Instead, the **oo::define** command will have a new subcommand, **constructor**, which will be used as follows:

```
oo::define myclass constructor {} {
    # body
}
```

The constructor so defined will act almost exactly like an instproc; it may call superclass constructors using the "super" command, etc. However, it may never be called explicitly, but only via the class's "create" and "new" methods.

Destructor Syntax

In XOTcl, a class's destructor is defined by overriding the "destroy" instproc. This is problematic for two reasons: first, a destructor doesn't need an argument list. An instproc is too powerful for the task. Second, successful destruction should not depend on the destructor's chaining to its superclass destructors properly.

Therefore, the class destructor will be defined by a new subcommand of **oo::define**, **destructor**, as follows:

```
oo::define myclass destructor {
    # Body
}
```

The destructor has no argument list.

The destructor cannot be called explicitly. Instead, the destructors are invoked in the proper order by the standard **destroy** method (defined in **oo::object**), which need never be overridden.

If an error occurs in a destructor, it will *not* prevent the object from being deleted. There is no guarantee to run destructors when an interpreter or Tcl-enabled process exits.

Behavior and Syntax of next

In XOTcl, the **next** command is used to invoke the "next" method in the dispatch chain. It optionally takes arguments to process (if they are omitted, it passes all arguments that were passed to the current method) and it calls the appropriate superclass implementation. But it does so without adjusting the Tcl stack, which forces classes to take extreme care when implementing code that needs to access variables or evaluate scripts in the scope of the code that invoked the method in the first place.

Therefore, the **next** command will perform Tcl stack management so that using **uplevel** and **upvar** in a method will be just like doing so in a normal procedure, no matter how the class containing that method is subclassed.

In addition, the adoption of TIP 157 makes explicit handling of arguments practical as code does not need to perform potentially troublesome operations with **eval**, and so **next** will always require that all argument be passed explicitly. This also makes it easier to decide to pass no arguments to a superclass implementation.

Desirable Changes

The changes described in this section are not absolutely essential to meeting the goals described earlier. However, they are desirable in that they lead to cleaner, more maintainable code.

Class vs. Object Method Naming

XOTcl has many features which can be applied to a class for use by all class instances, or to a single object. For example, a "filter" can be defined for a single object, while an "instfilter" can be defined for a class and applied to all instances of that class.

This is exactly backward. Most behavior will be defined for classes; additional per-object behavior is the special case, and consequently should have the less convenient name.

Therefore, all definition subcommands that begin with "inst" will be defined, in the core OO system without their "inst" prefix; the per-object subcommands will be manipulated via **oo::objdefine** or through the "self" prefix command (described above), to indicate that it is operating on the object itself and not the members of the class. Thus, a filter is defined on a class for its instances using the "filter" subcommand; a filter is defined on a particular object using the "self filter" subcommand (actually a subcommand of a subcommand).

```
oo::define someCls {
    method foo {} {...}
    self {
        method bar {args} {...}
        filter bar
    }
}
```

Procs vs. Methods

The word "proc" conveys a standalone function; an object's subcommands are more typically described as its "methods".

Therefore, the core OO system will use "method" in place of "proc" for definitions.

Public Names

In XOTcl, the main objects are *xotcl::Class* and *xotcl::Object*. However, the Tcl Style Guide dictates that public command names begin with a lower-case letter.

Therefore, all public names in the *oo::* namespace (i.e. the standard classes) will begin with a lower case letter, e.g., the standard core object system equivalents of *xotcl::Class* and *xotcl::Object* will be **oo::class** and **oo::object**.

This does not constrain any code making use of the OO system from naming objects however it wants.

API Specification

This section documents the core object system API in detail, based on the essential and desirable changes discussed in the previous sections.

Helper Commands

The namespace(s) that define the following three commands are not defined in this specification unless otherwise stated; all that is defined is that they will be on the object's **namespace path** during the execution of any method and should always be used without qualification.

my

The **my** command allows methods of the current object to be called during the execution of a method, just as if they were invoked using the object's command. Unlike the object's command, the **my** command may also invoke non-exported methods.

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

Note that each object has its own **my** command; they are all distinct from each other. This means that it is suitable for use for things like invoking callbacks (from general Tcl code) that are non-public methods. In particular, the use of **namespace code** for encapsulating the use of **my** for invoking unexported callback methods by non-object code is supported.

Note that the **my** command does not represent the name of the object.

next

The **next** command allows methods to invoke the implementation of the method with the same name in their superclass (as determined by the normal inheritance rules; if a per-object method overrides a method defined by the object's class, then the **next** command inside the object's method implementation will invoke the class's implementation of the method). The arguments to the **next** command are the arguments to be passed to the superclass method. The current stack level is temporarily bypassed for the duration of the processing of the **next** command; this allows a method to always execute identically with respect to the main calling context without needing to use some form of introspection to determine where that context is on the call frame stack (with a side effect of isolating method implementations from each other).

> **next** ?*arg arg ...*?

It is an error to invoke the **next** command when there is no superclass definition of the current method.

self

The **self** command allows executing methods to discover information about the object which they are currently executing in; it's always an error if not inside a method. Without arguments, the **self** command returns the current fully-qualified name of the object (to promote backward compatability). Otherwise, it is a command in the form of an ensemble (though it is not defined whether it is manipulable with **namespace ensemble**).

The following subcommands of **self** are defined. None of these subcommands take additional arguments.

caller: Returns a three-item list describing the class, object and method that invoked the current method, respectively. The syntax is as follows:

self caller

class: Returns the name of the class that defines the currently executing method. If the method was declared in the object instead of in the class, this returns the class of the object containing the method definition. The syntax is as follows:

self class

filter: When invoked inside a filter, returns a three-item list describing the object or class for which the filter has been registered. The first element is the name of the class or object, the second element is either **class** (for a filter defined on a class for its instances) or **object** (for a filter defined on a single object), and the third element is the name of the method. The syntax is as follows:

self filter

method: Returns the name of the currently executing method. The syntax is as follows:

self method

namespace: Returns the namespace associated with the current object. The syntax is as follows:

self namespace

next: Returns a two-element list describing the method that will be executed when the **next** command is invoked, or an empty list if there is no subsequent definition for the method. The first element of the list is the name of the object or class that contains the method, and the second element of the list is the name of the method. The syntax is as follows:

self next

object: Returns the name of the current object, the same as if the **self** command is invoked with no arguments. The syntax is as follows:

self object

target: When invoked from a filter, returns a two-item list consisting of the name of the class that holds the target method and the name of the target method. The syntax is as follows:

self target

For all these commands, when the name of a method is returned, it will be "<*constructor*>" when the method is a constructor, and "<*destructor*>" when the method is a destructor. It should be noted that these are not the actual names of the constructor and destructor (they are unnamed methods); they are just notational conventions supported by the **self** command.

The oo::define Command

oo::define *class subcommand* ?*arg* ...?

oo::define *class script*

oo::objdefine *object subcommand* ?*arg* ...?

oo::objdefine *object script*

The **oo::define** command is used to add behavior to classes, and the **oo::objdefine** command is used to add behavior to objects. The first form of each command is conventional for ensemble-like commands, except that the *class* or *object* argument precedes the *subcommand* argument. In the second form of each command, *script* is a Tcl script whose commands are the subcommands of **oo::define** or **oo::objdefine**; this is a notational convenience, as the two forms are semantically equivalent in what their capabilities are. (Note that the context in which *script* executes is otherwise not defined.)

Class-related Subcommands

The subcommands of **oo::define** (which may be unambiguously abbreviated in both the subcommand form and the script form) shall be:

- **constructor** - this takes two arguments (a **proc**-style argument list, and a body script), and sets the constructor for the instances of the class to be executed as defined by the body script after binding the actual arguments to the call that creates an instance of the class to the formal arguments listed. The constructor is called after the object is created but before any instance variables are guaranteed to be set. If no constructor is specified, the constructor will accept exactly the same arguments as the constructor in the parent class, and will delegate all the arguments to that parent-class constructor. The syntax is as follows:

  oo::define *class* **constructor** *argList body*

  Note that constructors of class mixins are also called, but constructors of object mixins are never called (as the object must exist before it can have an auxiliary class mixed into it).

- **destructor** - this defines the class destructor; a destructor is like a method but takes no arguments. Destructors are called on all classes that define them when the object is deleted, including classes that have been mixed in. The syntax is as follows:

  oo::define *class* **destructor** *body*

  Note that destructors *should always* use the **next** command within their implementation so that destructors of parent classes are also executed.

  Note also that destructors are called whenever the object is deleted by any mechanism (except when the overall interpreter is deleted, when execution of Tcl scripts has ceased to be possible anyway).

- **export** - this specifies that the named methods are exported, i.e., part of the public API of the class's instances. The syntax is as follows:

  oo::define *class* **export** *name* ?*name* ...?

  An exported method is accessible to clients of the class's instances; an unexported method is accessible only to the instances' own code through the **my** command.

- **filter** - this subcommand controls the list of filter methods for a class. Each filter method in the list is called when any method is invoked on the class's instances, and it is up to the filter to decide whether to invoke the filtered method call (using the **next** command) or return a suitable replacement value. The syntax is as follows:

  oo::define *class* **filter** ?*filterName filterName* ...?

- **forward** - this subcommand defines a class method which is automatically forwarded (i.e. delegated) to some other command, according to a simple pattern. Each *arg* is used literally. The syntax is as follows:

  oo::define *class* **forward** *name targetCmd* ?*arg* ...?

- **method** - this subcommand (only valid for classes) defines a class method (i.e. a method supported by every instance of the class). By default, methods are exported if they start with a lower-case letter (i.e., any character in \u0061 to \u007a inclusive) and are not exported otherwise. The syntax is as follows:

  oo::define *class* **method** *name args body*

- **mixin** - This subcommand defines a mixin for a class which is a way of bringing in additional method implementations (which may add to or wrap existing methods) on an *ad hoc* basis. The list of mixins is traversed when searching for methods before the inheritance hierarchy, and mixed-in methods may chain to any methods they override using the **next** command. The syntax is as follows:

  oo::define *class* **mixin** ?*mixinClass mixinClass* ...?

- **self** - This subcommand, which has the same syntax patterns as **oo::objdefine**, allows the manipulation of the class as an object. See **oo::objdefine** below for a description of the list of subcommands of **self**. The syntaxes are as follows:

  oo::define *class* **self** *subcommand* ?*arg* ...?

  oo::define *class* **self** *script*

- **superclass** - This specifies the superclass (or classes) of a class. Note that objects are always either classes or not classes, and cannot be changed from one to the other by any mechanism. The syntax is as follows:

  oo::define *class* **superclass** *classList*

- **unexport** - This specifies that the named methods are unexported, i.e., private. The syntax is as follows:

oo::define *class* **unexport** *name* ?*name ...*?

An exported method is accessible to clients of the object; an unexported method is accessible only to the object's own code, through the **my** command.

The following utility subcommands are also supported:

- **deletemethod** - This deletes one or more methods from a class; it doesn't modify any definitions of the method in superclasses, subclassses, instances or mixins. The method names must be specified exactly. Syntax is as follows:

  oo::define *class* **deletemethod** *name* ?*name ...*?

- **renamemethod** - This renames a method in a class from one thing to another; it doesn't modify any definitions of the method in superclasses, subclassses, instances or mixins. Syntax is as follows:

  oo::define *class* **renamemethod** *fromName toName*

Per-Object Subcommands

The following subcommands are all per-object versions of the class subcommands listed above. When they are applied to a class, they operate on the class instance itself as an object, and not on the instances (current and future) of that class (which is why the distinction is required).

- **class** - This subcommand gets and sets the class of an object. Changing the class of an object can result in many methods getting added or removed. Objects may not be changed between being class-objects and and non-class objects. The syntax is as follows:

  oo::objdefine *object* **class** *className*

  Note that when the class is changed of an object, no methods are called on that object, or on either the source or target classes, to indicate that the change has been carried out. This is up to the caller of the **class** subcommand.

- **deletemethod** - This is a per-object version of the **deletemethod** subcommand of **oo::define**, to which it is syntactically identical.
- **export** - This is a per-object version of the **export** subcommand of **oo::define**, to which it is syntactically identical.
- **filter** - This is a per-object version of the **filter** subcommand of **oo::define**, to which it is syntactically identical.
- **forward** - This is a per-object version of the **forward** subcommand of **oo::define**, to which it is syntactically identical.
- **method** - This is a per-object version of the **method** subcommand of **oo::define**, to which it is syntactically identical.
- **mixin** - This is a per-object version of the **mixin** subcommand of **oo::define**, to which it is syntactically identical.

- **renamemethod** - This is a per-object version of the **renamemethod** subcommand of **oo::define**, to which it is syntactically identical.
- **unexport** - This is a per-object version of the **unexport** subcommand of **oo::define**, to which it is syntactically identical.

The oo::copy Command

The **oo::copy** command creates an exact copy of an object with the given name. If *newName* is the empty string or unspecified, a new name will be generated automatically. The syntax is as follows:

> **oo::copy** *object* ?*newName*?

Note that this command does *not* copy the backing namespace, and nor does it execute any constructors. It is therefore up to the caller to copy such internal state of the object in the manner suitable for the object and its class tree; it is suggested that this be done by wrapping the **oo::copy** command in another command that defines which method is called.

Core Objects

The following classes are defined, and are the only pre-constructed objects in the core system.

oo::object

The root of the class hierarchy is **oo::object**. There are two ways to create a new instance of an object.

> **oo::object create** *name*
>
> **set** *var* **[oo::object new]**

The first constructs a new object called *name* of class *oo::object*; the object is represented as a command in the current scope. The second constructs a new object of class **oo::object** with a name guaranteed to be different from every existing command and returns the fully qualified of the command created (which it is naturally a good idea to save in a variable, perhaps called *var*).

The name of an object is also the name of a command in the form of an ensemble where the subcommands of the ensemble are the *exported* method names of the object. The command is not manipulable with **namespace ensemble**, but may be renamed.

The new object has one predefined exported method (**destroy**) and four predefined non-exported methods (**eval**, **unknown**, **variable** and **varname**). Other subcommands and other behaviour can be added using **oo::define**.

The **oo::object** class (an instance of **oo::class**) serves as the base class for all other core OO system classes.

Constructor and Destructor

The constructor for the **oo::object** class takes no arguments and does nothing. (The actual construction of an object is special and happens before any constructors are called.)

The destructor does nothing. (The actual destruction of an object is special and happens after all destructors have completed.)

Methods

The instances of **oo::object** (i.e. all objects and classes) have the following methods:

eval: This non-exported method concatenates its arguments according to the rules of **concat**, and evaluates the resulting script in the namespace associated with the object. The result of the script evaluation is the result of the *object* **eval** method. The syntax is as follows:

> *object* **eval** ?*arg arg ...*?

destroy: This exported method deletes the object; it takes no additional arguments and returns the empty string as its result. The syntax is as follows:

> *object* **destroy**

unknown: This non-exported method takes a method name and an arbitrary number of extra arguments and handles the absence of a method with the given name. The default implementation just generates a suitable error message that explains what commands are available given how the caller attempted to invoke the method, and ignores all the additional arguments. The syntax is as follows:

> *object* **unknown** *methodName* ?*arg ...*?
>
> Note that this method is not normally invoked directly.

variable: This non-exported method takes an arbitrary number of *unqualified* variable names and binds the variable with that name in the object's namespace to the same name in the current scope, provided the current scope is the body of a procedure, procedure-like method, or lambda term (as used with **apply**); if executed in a context where the current scope does not admit local variables, this method will have no effect. The syntax is as follows:

> *object* **variable** ?*varName varName ...*?
>
> However, it will be more commonly used as:
>
> **my variable** ?*varName varName ...*?
>
> Each *varName* argument is the name of a variable in the namespace associated with the object, and must not contain any namespace separators. Each named variable will be bound to a local variable in the current scope with the same name.

varname: This non-exported method takes one argument, the name of a variable to be resolved in the context of the object's namespace, and returns the fully qualified name of the variable such that it can be used with the **vwait** command or extensions such as Tk (e.g., for the **label** widget's **-textvariable** option). This method does not assign any value to the variable. The syntax is as follows:

> *object* **varname** *varName*

However, it will be more commonly used as:

my varname *varName*

Unknown Method Handling

When an attempt is made to invoke an unknown method on any object, the core then attempts to pass *all* the arguments (including the method name) to the **unknown** method of the object. The default implementation of the **unknown** method is specified by the **oo::object** class, and just generates a suitable "unknown subcommand" error message.

oo::class

This class is the class of all classes (i.e. its instances are objects that manufacture objects according to a standard pattern). Note that **oo::object** is an instance of **oo::class**, as is **oo::class** itself.

oo::class create *name* ?*definition*?

This creates a new class called *name*; the class is an object in its own right (of class **oo::class**), and hence is represented as a command in the current scope. **oo::class** returns the fully qualified command name.

The newly-created class command is used to define objects which belong to the class, just as **oo::object** is. By default, instances of the new class have no more behaviour than instances of **oo::object** do; new class behavior can be added to the class in two ways. First, a *definition* can be specified when creating the class; second, additional behaviour can be added to the class using **oo::define**.

The definition, if given, consists of a series of statements that map to the subcommands of **oo::define**. The following three code snippets are equivalent; each defines a class called **::dog** whose instances will have two subcommands: **bark** and **chase**.

```
# Method 1
oo::class create dog

oo::define dog method bark {} {
    puts "Woof, woof!"
}

oo::define dog method chase {thing} {
    puts "Chase $thing!"
}

# Method 2
oo::class create dog

oo::define dog {
    method bark {} {
        puts "Woof, woof!"
    }

    method chase {thing} {
        puts "Chase $thing!"
    }
}
```

```
# Method 3
oo::class create dog {
    method bark {} {
        puts "Woof, woof!"
    }

    method chase {thing} {
        puts "Chase $thing!"
    }
}
```

Constructor and Destructor

The constructor for **oo::class** concatenates its arguments and passes the resulting script to **oo::define** (along with the fully-qualified name of the created class, of course).

Classes have no destructor by default. (Actual class destruction is special, and happens after all destructors have been executed.)

Methods

The instances of **oo::class** have the following methods:

create: Creates a new instance of the class with the given name. All subsequent arguments are given to the class's constructor (and so must actually match the syntax pattern specified in the constructor definition). The result of the **create** method is always the fully-qualified name of the newly-created object. The syntax is as follows:

> *class* **create** *objName* ?*arg arg* ...?

new: Creates a new instance of the class with an automatically chosen name. All subsequent arguments are given to the class's constructor (and so must actually match the syntax pattern specified in the constructor definition). The result of the **new** method is always the fully-qualified name of the newly-created object. The syntax is as follows:

> *class* **new** ?*arg arg* ...?
>
> Note that the *oo::class* object itself does not export the **new** method; it is good practice for all classes to have names.

createWithNamespace: Creates a new instance of a class with a given name and a given name of backing namespace. This method (required to provide proper support for putting [incr Tcl] on top of the core OO system) is not exported by default. Apart from the *nsName* parameter, it is the same as the **create** method.

> *class* **createWithNamespace** *objName nsName* ?*arg arg* ...?

Introspection Support

The core Tcl **info** command shall be extended in the following ways.

An [info object] Subcommand

An **object** subcommand that shall provide information about a particular object. Its first argument shall be the name of an object to get information about, its second argument shall be a subsubcommand indicating the type of information to retrieve and all subsequent arguments shall be arguments, as appropriate. The following types of information shall be available:

class: Returns the class of an object, or if *className* is specified, whether the object is (directly or indirectly through inheritance or mixin) an instance of the named class.

> **info object class** *object* ?*className*?

definition: Returns the formal argument list and body used to define a method.

> **info object definition** *object method*

filters: Returns the list of filters defined for an object.

> **info object filters** *object*

forward: Returns the list of words that form the command prefix that a method is forwarded to.

> **info object forward** *object method*

isa: Returns boolean information about how an object relates to the class hierarchy. Supports a range of subcommands to allow the specification of what sort of test is to be performed:

> class: Returns whether the named object is a class.
>
> > **info object isa class** *object*
>
> metaclass: Returns whether the named object is a class that can create other classes (i.e. is **oo::class** or one of its subclasses).
>
> > **info object isa metaclass** *object*
>
> mixin: Returns whether the named object has *mixinClassName* as one of its mixins.
>
> > **info object isa mixin** *object mixinClassName*
>
> object: Returns whether *object* really names an object.
>
> > **info object isa object** *object*
>
> typeof: Returns whether the object is of type *class* (i.e. an instance of that class or an instance of a subclass of that class).
>
> > **info object isa typeof** *object class*

methods: Returns the list of methods defined for an object. Supports the options **-all** to also look at the class hierarchy for the object, and **-private** to get the list of methods supported by **my** for the object.

info object methods *object options*

mixins: Returns the list of mixins for an object.

info object mixins *object*

vars: Returns the list of all variables defined within the object, or optionally just those that match *pattern* according to the rules of **string match**.

info object vars *object* ?*pattern*?

An [info class] Subcommand

A **class** subcommand that shall provide information about a particular class. Its first argument shall be the name of a class to get information about, its second argument shall be a subsubcommand indicating the type of information to retrieve and all subsequent arguments shall be arguments, as appropriate. The following types of information shall be available:

constructor: Returns the formal argument list and body used to define the constructor, or an empty list if no constructor is present.

info class constructor *class*

definition: Returns the formal argument list and body used to define a method.

info class definition *class method*

destructor: Returns the body of the destructor, or an empty string if no destructor is present.

info class destructor *class*

filters: Returns the list of filters defined for a class.

info class filters *class*

forward: Returns the list of words that form the command prefix that a method is forwarded to.

info class forward *class method*

instances: Returns a list of all direct instances of the class (but not instances of any subclasses of the class), or optionally just those that match *pattern* according to the rules of **string match**.

info class instances *class* ?*pattern*?

methods: Returns the list of methods defined by a class. Supports the options **-all** to also look at the class hierarchy, and **-private** to get the list of methods supported by **my** for the object's instances.

info class methods *class options*

subclasses: Returns a list of all subclasses of the class, or optionally just those that match *pattern* according to the rules of **string match**.

info class subclasses *class* ?*pattern*?

superclasses: Returns a list of all superclasses of the named class in the class hierarchy. The list will be ordered in inheritance-precedence order.

> **info class superclasses** *class*

Extending the Introspection Capabilities

Other forms of introspection subcommands may be added to **info object** and **info class** by creating exported commands in the namespaces **oo::InfoObject** and **oo::InfoClass** respectively.

Issues with Objects and Namespaces

Every object has a distinct namespace associated with it, the name of which is outside the scope of this specification. It is the name of this namespace that is returned by **self namespace**.

The namespace does have a path set, as if by calling **namespace path**; this is how the **next** and **self** commands are provided, though since they are never usable outside the scope of the body of a method, the namespace which they originate from is out of the scope of this specification.

The **my** command is the only command in the object's namespace by default (i.e., this command is truly per-object). It is not exported from the object's namespace, nor are any other commands exported from or imported into the namespace.

Methods are not commands, and so completely ignore the **namespace export** command, nor does **namespace unknown** get involved at any point during the location of a method. (They may run during the processing of a method body; it is a context very similar to a normal procedure body.) Similarly, a method may not be **namespace import**ed from another namespace.

Each method (including both the constructor and destructor) executed by an object executes in that object's namespace. Changes made by a method to the namespace (including both command declaration and uses of **namespace import**) will be seen by all other methods invoked on the same object. Note that methods declared by classes still execute in the instance objects' namespaces.

C API

Note: This API is probably incomplete. Future TIPs may extend or completely revise it.

Datatypes

The following public datatypes shall be declared in tcl.h:

Tcl_Object: An opaque handle to an object.

Tcl_Class: An opaque handle to a class.

Tcl_Method: An opaque handle to a method.

Tcl_ObjectContext: An opaque handle to an object method call context.

Tcl_MethodType: A structure describing the type of a method implementation. It shall have the following fields:

version: The version number of the structure, which should always be referred to as TCL_OO_METHOD_VERSION_CURRENT in source code (currently ignored, but allows transparent versioning in the future).

name: The name of the method type, for debugging.

callProc: A pointer to a function that defines how to call method implementations of this type. Must not be NULL.

deleteProc: A pointer to a function that defines how to delete the *clientData* associated with a particular method implementation instance. If NULL, no deletion of the *clientData* is required.

cloneProc: A pointer to a function that defines how to copy the *clientData* associated with a particular method implementation instance during the copying of an object or class with **oo::copy**. If NULL, the method will be cloned by just copying the *clientData*.

Tcl_ObjectMetadataType: A structure describing the type of some arbitrary non-NULL metadata attached to an object or class. It shall have the following fields:

version: The version number of the structure, which should always be referred to as TCL_OO_METADATA_VERSION_CURRENT in source code (currently ignored, but allows for transparent versioning in the future).

name: The name of the metadata type, for debugging.

deleteProc: A pointer to a function that defines how to delete some metadata associated with this type. Must not be NULL.

cloneProc: A pointer to a function that defines how to copy some metadata associated with this type during the copying of an object or class with **oo::copy**. If NULL, the metadata will not be copied. (*Open issue:* There is a use case for making objects unclonable; consider the case where the metadata consists of one or more OS resource handles. Simply shallow-copying resource handles is a bad idea, but deep-copying them may well be infeasible. Not all objects can handle copy-on-write semantics gracefully.)

Tcl_MethodCallProc: The type of the *callProc* field of the Tcl_MethodType structure. It is a pointer to a function that is used to implement how a method implementation is called. It takes five arguments and returns a normal Tcl result code. The arguments are:

clientData: Some method implementation instance specific data. Note that this is specific to the instance of the method implementation, and not (necessarily) the instance of the object.

interp: The Tcl interpreter reference.

objectContext: The object method call context, through which useful information (such as what object this method was invoked upon) can be obtained.

objc: The number of arguments.

> objv: The actual list of arguments. Since the number of arguments required to indicate the method may vary, the method implementation should look up how many to skip over using the object method call context.

Tcl_MethodDeleteProc: The type of the *deleteProc* field of the Tcl_MethodType structure. It is a pointer to a function that is used to delete *clientData* values associated with a method instance. It takes a single argument (the *clientData* to delete) and has no return value.

Tcl_MethodCloneProc: The type of the *cloneProc* field of the Tcl_MethodType structure. It is a pointer to a function that is used to make copies of *clientData* values associated with a method instance suitable for use in another method instance. It takes two arguments (the *clientData* to clone, and a pointer to a variable into which to write the cloned *clientData*) and returns either TCL_OK or TCL_ERROR, with the method only being cloned if the result is TCL_OK (the method is silently not cloned otherwise).

Tcl_ObjectMapMethodNameProc: The type of a callback function used to adjust the mapping of objects and method names to implementations, which is required to support building [incr Tcl] on top of the core OO system. It takes four arguments, being the interpreter, the object that the method is being invoked upon, a point to a variable to contain the class in the hierarchy to start the search for components of the method chain from, and an unshared object holding the method name as supplied and which can be modified if the method to look for is not the literal name passed in.

> This is necessary because [incr Tcl] allows the invoking of superclass implementations of a method using a syntax like "*superclassName*::*methodName*" where *superclassName* may name any superclass of the current class, and *methodName* may be any method name.
>
> Note that the exact definition of this type is subject to change at the moment in order to ensure that it can be connected to the method dispatch engine efficiently. The type will be finalized before the release of Tcl 8.6.

Tcl_ObjectMetadataDeleteProc: The type of the *deleteProc* field of the Tcl_ObjectMetadataType structure. It is a pointer to a function that is used to delete *metadata* values attached to an object or a class. It takes a single argument (the *metadata* to delete) and has no return value.

Tcl_ObjectMetadataCloneProc: The type of the *cloneProc* field of the Tcl_ObjectMetadataType structure. It is a pointer to a function that is used to create a copy of *metadata* values attached to an object or a class. It takes three argument, (the interpreter, the *metadata* to copy, and a pointer to a variable into which to write the copy, or NULL if the copy is not to be performed) and returns a standard Tcl result code.

Functions

The following functional operations are defined:

Tcl_NewMethod: This function creates a new method on a class (and hence on all instances of that class). It has the following signature:

> Tcl_Method **Tcl_NewMethod**(Tcl_Interp *interp, Tcl_Class *cls*, Tcl_Obj **nameObj*, int *isPublic*, const Tcl_MethodType **typePtr*, ClientData *clientData*)

> If the method is created with a NULL *nameObj*, it must be installed manually into the class as a constructor or destructor (in the latter case, it is important that the method be able to execute without additional arguments). Note that a NULL *typePtr* is reserved for internal use.

Tcl_ClassSetConstructor: This function installs a method into a class as a constructor for instances of that class. The method must have been created with **Tcl_NewMethod** with a NULL *nameObj* argument. It has the following signature:

> void **Tcl_ClassSetConstructor**(Tcl_Class *cls*, Tcl_Method *method*)

Tcl_ClassSetDestructor: This function installs a method into a class as a destructor for instances of that class. The method must have been created with **Tcl_NewMethod** with a NULL *nameObj* argument. It has the following signature:

> void **Tcl_ClassSetConstructor**(Tcl_Class *cls*, Tcl_Method *method*)

Tcl_NewInstanceMethod: This function creates a new method on an object. It has the following signature:

> Tcl_Method **Tcl_NewInstanceMethod**(Tcl_Interp **interp*, Tcl_Object *object*, Tcl_Obj **nameObj*, int *isPublic*, const Tcl_MethodType **typePtr*, ClientData *clientData*)
>
> Note that a NULL *typePtr* is reserved for internal use, and *nameObj* must not be NULL.

Tcl_NewObjectInstance: This function creates a new instance of a class, calling any defined constructors. It has the following signature, returning NULL (and setting a message in the interpreter) if the object creation failed:

> Tcl_Object **Tcl_NewObjectInstance**(Tcl_Interp **interp*, Tcl_Class *cls*, const char **name*, const char **nsName*, int *objc*, Tcl_Obj *const **objv*, int *skip*)
>
> Both *name* and *nsName* may be NULL, in which case the constructor code picks a default that doesn't clash with any previously existing commands or namespaces.

Tcl_CopyObjectInstance: This function creates a copy of an object (including classes) without copying the backing namespace or executing any constructors. It has the following signature, returning NULL (and setting a message in the interpreter) if the object copying failed:

> Tcl_Object **Tcl_CopyObjectInstance**(Tcl_Interp **interp*, Tcl_Object *sourceObject*, const char **targetName*)
>
> Note that the copying of an object can fail if the copy code for one of the metadata coping functions fails, so those functions can veto copying. Also note that if *targetName* is NULL, a name will be picked for the object. Currently no control over the naming of the target object's namespace is provided.

Tcl_GetObjectFromObj: This function converts from a Tcl_Obj holding the name of an object to a Tcl_Object handle. It returns NULL (leaving an error message in the interpreter) if the conversion fails.

> Tcl_Object **Tcl_GetObjectFromObj**(Tcl_Interp **interp*, Tcl_Obj **objPtr*)

Tcl_ObjectContextInvokeNext: This function invokes the next method implementation in a method call chain, and is the internal implementation of the **next** command. It has the following signature:

> int **Tcl_ObjectContextInvokeNext**(Tcl_Interp **interp*, Tcl_ObjectContext *context*, int *objc*, Tcl_Obj *const **objv*, int *skip*)

Tcl_ClassGetMetadata: This function retrieves the metadata attached to the class *cls* that is associated with the type *typePtr*. It returns NULL if no data of that type is attached.

> ClientData **Tcl_ClassGetMetadata**(Tcl_Class *cls*, const Tcl_ObjectMetadataType **typePtr*)

Tcl_ClassSetMetadata: This function attaches metadata, *metadata*, of a specific type, *typePtr*, to the class, *clazz*, or removes the metadata of that type if *metadata* is NULL. It is a no-op to remove metadata of a type that is not attached in the first place.

> void **Tcl_ClassSetMetadata**(Tcl_Class *clazz*, const Tcl_ObjectMetadataType **typePtr*, ClientData *metadata*)

Tcl_ObjectGetMetadata: This function retrieves the metadata attached to the object *object* that is associated with the type *typePtr*. It returns NULL if no data of that type is attached.

> ClientData **Tcl_ObjectGetMetadata**(Tcl_Object *object*, const Tcl_ObjectMetadataType **typePtr*)

Tcl_ObjectSetMetadata: This function attaches metadata, *metadata*, of a specific type, *typePtr*, to the object, *object*, or removes the metadata of that type if *metadata* is NULL. It is a no-op to remove metadata of a type that is not attached in the first place.

> void **Tcl_ObjectSetMetadata**(Tcl_Object *object*, const Tcl_ObjectMetadataType **typePtr*, ClientData *metadata*)

Tcl_ObjectGetMethodNameMapper: This function retrieves the current method name mapping function for an object, or NULL if none was set. It has the following signature:

> Tcl_ObjectMapMethodNameProc **Tcl_ObjectGetMethodNameMapper**(Tcl_Object *object*)

Tcl_ObjectSetMethodNameMapper: This functionsets the method name mapping function for an object, or removes it if the function is set to NULL. It has the following signature:

> void **Tcl_ObjectSetMethodNameMapper**(Tcl_Object *object*, Tcl_ObjectMapMethodNameProc *methodNameMapper*)

The following pure inspective (i.e., non-state changing) operations are defined:

Tcl_GetClassAsObject: This gets the object that represents a class.

> Tcl_Object **Tcl_GetClassAsObject**(Tcl_Class *clazz*)

Tcl_GetObjectAsClass: This gets the class that an object represents (or NULL if the object does not represent a class).

> Tcl_Class **Tcl_GetObjectAsClass**(Tcl_Object *object*)

Tcl_GetObjectCommand: This gets the command for an object. It is the name of this command that represents the object at the script level, and as such, it may be renamed.

Tcl_Command **Tcl_GetObjectCommand**(Tcl_Object *object*)

Tcl_GetObjectNamespace: This gets the object's private namespace.

Tcl_Namespace **Tcl_GetObjectNamespace**(Tcl_Object *object*)

Tcl_MethodDeclarerClass: This gets the class that declared a method, or NULL if the method is a per-object method.

Tcl_Class **Tcl_MethodDeclarerClass**(Tcl_Method *method*)

Tcl_MethodDeclarerObject: This gets the object that declared a method, or NULL if the method is a class method.

Tcl_Object **Tcl_MethodDeclarerObject**(Tcl_Method *method*)

Tcl_MethodIsPublic: This returns whether a method is a public method. This status might be overridden in subclasses or objects.

int **Tcl_MethodIsPublic**(Tcl_Method *method*)

Tcl_MethodIsType: This returns whether a method is a specific type of method, and if so, also returns the *clientData* for the type. No way of inspecting method types for which you do not have a pointer to the type structure is provided.

int **Tcl_MethodIsType**(Tcl_Method *method*, const Tcl_MethodType **typePtr*,
ClientData **clientDataPtr*)

Tcl_MethodName: This returns the name of a method.

Tcl_Obj **Tcl_MethodName**(Tcl_Method *method*)

Tcl_ObjectDeleted: This returns whether an object has been deleted (assuming deletion has not yet completed, i.e., that the destructor is currently being processed).

int **Tcl_ObjectDeleted**(Tcl_Object *object*)

Tcl_ObjectContextIsFiltering: This returns whether the method call context is working with a filter or not.

int **Tcl_ObjectContextIsFiltering**(Tcl_ObjectContext *context*)

Tcl_ObjectContextMethod: This returns the method call context's current method instance.

Tcl_Method **Tcl_ObjectContextMethod**(Tcl_ObjectContext *context*)

Tcl_ObjectContextObject: This returns the method call context's object (i.e., the object which was invoked).

Tcl_Object **Tcl_ObjectContextObject**(Tcl_ObjectContext *context*)

Tcl_ObjectContextSkippedArgs: This returns the number of arguments to be skipped (this varies because the method instance may be invoked through either [obj method ...] or through [next ...]).

int **Tcl_ObjectContextSkippedArgs**(Tcl_ObjectContext *context*)

Not Addressed in this Document

This TIP does not address the reqirements for management of variables on a class level or for "class methods" (in the Java sense). These will need to be the subject of future TIPs.

Copyright

The following sections are non-normative.

Appendix: Class Hierarchy for Support of Other OO Systems

When using the OO system as a basis for some other object system, it is useful for all classes and objects to derive from some other object root for compatability with existing practice. To see how to do this, consider this class hierarchy (targetted at XOTcl) outlined below. The XOTcl *Object* class would derive from the core **oo::object** class, and the XOTcl *Class* class would derive from the core **oo::class** and the XOTcl *Object* classes. This would give the following diagram (core classes are in lower case with their namespace omitted, XOTcl classes are in upper case, with namespace omitted).

```
                 +--------+
        ,------->| object |
        |        +--------+
        |            ,^.
creates |       ______|______
        |      |             |
        |  +-------+    +--------+
        `--| class |    | Object |<-.
           +-------+    +--------+  |
              ,^.           ,^.     |
               |______ ______|      | creates
                      |             |
                  +-------+         |
                  | Class |---------'
                  +-------+
```

Note that **class** instances create **object**s (or subclasses thereof), but *Class* instances create _Object_s (or subclasses thereof).

Appendix: XOTcl Features Omitted from the Core OO System

Object Methods

Object::autoname: This is trivially implemented in a small procedure, and core objects can pick names for themselves and are renameable.

Object::check: Preconditions and postconditions are not supported (they add a lot of complexity) and neither are invariants. Hence, there is no need to control whether they are executed.

Object::cleanup: This is not an especially well-defined method (what if the object happens to hold handles to complex resources such as network sockets; it is not generally possible for the state of the remote server to be reset) and can be added in any compatability layer.

Object::configure: This feature has been deliberately omitted from the core object system. This would be value added by any XOTcl extension.

Object::extractConfigureArg: This feature is part of **configure**.

Object::getExitHandler: This feature is not necessary for this version. If it existed, it would not need to be a part of the base object.

Object::info: The introspection features are moved into the core **info** command.

Object::invar: Invariants may be implemented using filters.

Object::move: This feature is equivalent to the use of the standard **rename** operation.

Object::noinit: This feature has been deliberately omitted from the core object system because its use is dependent on the use of other deliberately-omitted features (i.e., **configure**). This would be value added by any XOTcl extension.

Object::parameter and Object::parametercmd: The core object system provides tools for doing parameters, but does not provide an implementation on the grounds that it is pretty easy to add.

Object::requireNamespace: Objects always have a namespace.

Object::setExitHandler: See the comments for **getExitHandler** above.

Class Methods

Class::__unknown: Auto-loading of unknown classes is handled by the standard core **unknown** mechanism.

Class::abstract: Abstractness is relatively easy to implement on top of the proposed infrastructure and is not critical to getting an implementation.

Class::allinstances: This feature is trivially implemented in a small procedure.

Class::alloc: The core objects have no default behaviour, so the difference with the basic core class behaviour is moot.

Class::create: Core object creation is a much more sealed process, but the lack of **configure**-like behaviour means that the complexity of this method is not necessary. Instead, constructors are called automatically.

Class::parameterclass: Core object system parameters are not implemented by classes.

Class::volatile: This feature is omitted as it is believed that it is possible to implement automated lifecycle management as a mixin.

Other Commands

getExitHandler, setExitHandler: Exit handlers are out of scope for the core object system.