OTcl - Object Tcl Extensions

This printer-friendly document is reformatted the content from
http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl/

The CMT project has adopted OTcl as the base for the CMT Media Playback API, but unfortunately found MIT Version 0.96 to be lacking a couple of needed features, as well as not being compatible with Tcl/Tk 8.0. We are therefore including our modified version of OTcl with CMT 4.0.

What is OTcl?

OTcl, short for MIT Object Tcl, is an extension to Tcl/Tk for object-oriented programming. It shouldn't be confused with the IXI Object Tcl extension by Dean Sheenan. Some of OTcl's features as compared to alternatives are:

- designed to be dynamically extensible, like Tcl, from the ground up
- builds on Tcl syntax and concepts rather than importing another language
- compact yet powerful object programming system (draws on CLOS, Smalltalk, and Self)
- fairly portable implementation (2000 lines of C, without core hacks)

For documentation about objects, classes, and their capabilities, see the following reference pages:

- Tutorial
- OTcl Objects
- OTcl Classes
- OTcl Autoloading
- OTcl C API

OTcl Tutorial (Version 0.96, September 95)

This tutorial is intended to start you programming in OTcl quickly, assuming you are already familiar with object-oriented programming. It omits many details of the language that can be found in the reference pages Objects in OTcl and Classes in OTcl. It also doesn't mention the C API or describe how to autoload classes.

Comparison with C++

To the C++ programmer, object-oriented programming in OTcl may feel unfamiliar at first. Here are some of the differences to help you orient yourself.

- Instead of a single class declaration in C++, write multiple definitions in OTcl. Each method definition (with `instproc`) adds a method to a class. Each instance variable definition (with `set` or via `instvar` in a method body) adds an instance variable to an object.
- Instead of a constructor in C++, write an `init` instproc in OTcl. Instead of a destructor in C++, write a `destroy` instproc in OTcl. Unlike constructors and destructors, init and destroy methods do not combine with base classes automatically. They should be combined explicitly with `next`.
- Unlike C++, OTcl methods are always called through the object. The name `self`, which is equivalent to `this` in C++, may be used inside method bodies. Unlike C++, OTcl methods are always virtual.
- Instead of calling shadowed methods by naming the method explicitly as in C++, call them with `next`. `next` searches further up the inheritance graph to find shadowed methods automatically. It allows methods to be combined without naming dependencies.
- Avoid using static methods and variables, since there is no exact analogue in OTcl. Place shared variables on the class object and access them from methods by using `$class`. This behavior will then be inherited. For inherited methods on classes, program with meta-classes. If inheritance is not needed, use `proc` methods on the class object.

Programming in OTcl

Suppose we need to work with many bagels in our application. We might start by creating a `Bagel` class.

```
% Class Bagel
Bagel
```

We can now create bagels and keep track of them using the `info` method.

```
% Bagel abagel
abagel
% abagel info class
```

```
Bagel
% Bagel info instances
abagel
```

Of course, bagels don't do much yet. They should remember whether they've been toasted. We can create and access an instance variable with the `set` method. All instance variables are public in the sense of C++. Again, the `info` method helps us keep track of things.

```
% abagel set toasted 0
0
% abagel info vars
toasted
% abagel set toasted
0
```

But we really want them to begin in an untoasted state to start with. We can achieve this by adding an `init` instproc to the `Bagel` class. Generally, whenever you want newly created objects to be initialized, you'll write an `init` instproc for their class.

```
% Bagel instproc init {args} {
  $self set toasted 0
  eval $self next $args
}
% Bagel bagel2
bagel2
% bagel2 info vars
toasted
% bagel2 set toasted
0
```

There are several things going on here. As part of creating objects, the system arranges for `init` to be called on them just after they are allocated. The `instproc` method added a method to the `Bagel` class for use by its instances. Since it is called init, the system found it and called it when a new bagel was created.

The body of the `init` instproc also has some interesting details. The call to `next` is typical for init methods, and has to do with combining all inherited init methods into an aggregate init. We'll discuss it more later. The variable called `self` is set when a method is invoked, and contains the name of the object on behalf of which it is running, or `bagel2` in this case. It's used to reach further methods on the object or inherited through the object's class, and is like `this` in C++. There are also two other special variables that you may be interested in, `proc` and `class`.

Our bagels now remember whether they've been toasted, except for the first one that was created before we wrote an init. Let's destroy it and start again.

```
% Bagel info instances
bagel2 abagel
% abagel destroy
```

```
% Bagel info instances
bagel2
% Bagel abagel
abagel
```

Now we're ready to add a method to bagels so that we can toast them. Methods stored on classes for use by their instances are called instprocs. They have an argument list and body like regular Tcl procs. Here's the toast instproc.

```
% Bagel instproc toast {} {
  $self instvar toasted
  incr toasted
  if {$toasted>1} then {
    error "something's burning!"
  }
  return {}
}
% Bagel info instprocs
init toast
```

Aside from setting the `toasted` variable, the body of the toast instproc demonstrates the `instvar` method. It is used to declare instance variables and bring them into local scope. The instance variable `toasted`, previously initialized with the `set` method, can now be manipulated through the local variable `toasted`.

We invoke the toast instproc on bagels in the same way we use the `info` and `destroy` instprocs that were provided by the system. That is, there is no distinction between user and system methods.

```
% abagel toast
% abagel toast
something's burning!
```

Now we can add spreads to the bagels and start tasting them. If we have bagels that aren't topped, as well as bagels that are, we may want to make toppable bagels a separate class. Let explore inheritance with these two classes, starting by making a new class `SpreadableBagel` that inherits from `Bagel`.

```
% Class SpreadableBagel -superclass Bagel
SpreadableBagel
% SpreadableBagel info superclass
Bagel
% SpreadableBagel info heritage
Bagel Object
```

More options on the `info` method let us determine that `SpreadableBagel` does indeed inherit from `Bagel`, and further that it also inherits from `Object`. `Object` embodies the basic functionality of all objects, from which new classes inherit by default. Thus `Bagel` inherits from `Object` directly (we didn't tell the system otherwise) while `SpreadableBagel` inherits from `Object` indirectly via `Bagel`.

The creation syntax, with its "-superclass", requires more explanation. First, you might be wondering why all methods except `create` are called by using their name after the object name, as the second argument. The answer is that `create` is called as part of the system's unknown mechanism if no other method can be found. This is done to provide the familiar widget-like creation syntax, but you may call `create` explicitly if you prefer.

Second, as part of object initialization, each pair of arguments is interpreted as a (dash-preceded) procedure name to invoke on the object with a corresponding argument. This initialization functionality is provided by the `init` instproc on the `Object` class, and is why the `Bagel init` instproc calls `next`. The following two code snippets are equivalent (except in terms of return value). The shorthand it what you use most of the time, the longhand explains the operation of the shorthand.

```
% Class SpreadableBagel
SpreadableBagel
% SpreadableBagel superclass Bagel
% Class create SpreadableBagel
SpreadableBagel
% SpreadableBagel superclass Bagel
% Class SpreadableBagel -superclass Bagel
SpreadableBagel
```

Once you understand this relationship, you will realize that there is nothing special about object creation. For example, you can add other options, such as one specifying the size of a bagel in bites.

```
% Bagel instproc size {n} {
  $self set bites $n
}
% SpreadableBagel abagel -size 12
abagel
% abagel set bites
12
```

We need to add methods to spread toppings to `SpreadableBagel`, along with a list of current toppings. If we wish to always start with an empty list of toppings, we will also need an `init` instproc.

```
% SpreadableBagel instproc init {args} {
  $self set toppings {}
  eval $self next $args
}
% SpreadableBagel instproc spread {args} {
  $self instvar toppings
  set toppings [concat $toppings $args]
  return $toppings
}
```

Now the use of `next` in the `init` method can be further explained. `SpreadableBagels` are also bagels, and need their `toasted` variable initialized to zero. The call to `next`

arranges for the next method up the inheritance tree to be found and invoked. It provides functionality similar to call-next-method in CLOS.

In this case, the `init` instproc on the `Bagel` class is found and invoked. Eval is being used only to flatten the argument list in `args`. When `next` is called again in `Bagels init` instproc, the `init` method on `Object` is found and invoked. It interprets its arguments as pairs of procedure name and argument values, calling each in turn, and providing the option initialization functionality of all objects. Forgetting to call `next` in an `init` instproc would result in no option initializations.

Let's add a taste instproc to bagels, splitting its functionality between the two classes and combining it with `next`.

```
% Bagel instproc taste {} {
  $self instvar toasted
  if {$toasted == 0} then {
    return raw!
  } elseif {$toasted == 1} then {
    return toasty
  } else {
    return burnt!
  }
}

% SpreadableBagel instproc taste {} {
  $self instvar toppings
  set t [$self next]
  foreach i $toppings {
    lappend t $i
  }
  return $t
}

% SpreadableBagel abagel
abagel
% abagel toast
% abagel spread jam
jam
% abagel taste
toasty jam
```

Of course, along come sesame, onion, poppy, and a host of other bagels, requiring us to expand our scheme. We could keep track of flavor with an instance variable, but this may not be appropriate. Flavor is an innate property of the bagels, and one that can affect other behavior - you wouldn't put jam on an onion bagel, would you? Instead of making a class heirarchy, let's use multiple inheritance to make the flavor classes mixins that add a their taste independent trait to bagels or whatever other food they are mixed with.

```
% Class Sesame
Sesame
% Sesame instproc taste {} {
  concat [$self next] "sesame"
```

```
}
% Class Onion
Onion
% Onion instproc taste {} {
  concat [$self next] "onion"
}
% Class Poppy
Poppy
% Poppy instproc taste {} {
  concat [$self next] "poppy"
}
```

Well, they don't appear to do much, but the use of `next` allows them to be freely mixed.

```
% Class SesameOnionBagel -superclass {Sesame Onion SpreadableBagel}
SesameOnionBagel
% SesameOnionBagel abagel -spread butter
% abagel taste
raw! butter onion sesame
```

For multiple inheritance, the system determines a linear inheritance ordering that respects all of the local superclass orderings. You can examine this ordering with an `info` option. `next` follows this ordering when it combines behavior.

```
% SesameOnionBagel info heritage
Sesame Onion SpreadableBagel Bagel Object
```

We can also combine our mixins with other classes, classes that need have nothing to do with bagels, leading to a family of chips.

```
% Class Chips
Chips
% Chips instproc taste {} {
  return "crunchy"
}
% Class OnionChips -superclass {Onion Chips}
OnionChips
% OnionChips abag
abag
% abag taste
crunchy onion
```

Other Directions

There are many other things we could do with bagels, but it's time to consult the reference pages. The OTcl language aims to provide you with the basic object-oriented programming features that you need for most tasks, while being extensible enough to allow you to customize existing features or create your own.

Here are several important areas that the tutorial hasn't discussed.

- There is support for autoloading libraries of classes and methods. See OTcl Autoloading for details.
- There is a C level interface (as defined by `otcl.h`) that allows new objects and classes to be created, and methods implemented in C to be added to objects. See OTcl C API for details.
- Classes are special kinds of objects, and have all of the properties of regular objects. Thus classes are a convenient repository for procedures and data that are shared by their instances. And the behavior of classes may be controlled by the standard inheritance mechanisms and the class `Class`.
- Methods called procs can be added to individual object, for sole use by that object. This allows particular objects to be hand-crafted, perhaps storing their associated procedures and data.
- User defined methods are treated in the same way as system provided methods (such as `set` and `info`). You can use the standard inheritance mechanisms to provide your own implementation in place of a system method.
- There are several other system methods that haven't been described. `array` gives information on array instance variables, `unset` removes instance variables, there are further info options, and so forth.

OTcl Objects

Overview

This reference page describes the functionality provided for all objects by methods on the `Object` class. See the tutorial for an introduction to programming in OTcl, and the C API page for details of manipulating objects from C.

Objects in OTcl are instances of classes. They are created through class objects either implicitly, with a widget command syntax, or explicitly, by calling a creation method. After defining a class Bagel with instprocs flavor and size below, a new bagel object called abagel is created and initialized by calling its flavor and size instprocs. The creation process may be customized for any class. See OTcl Classes for details.

```
% Class Bagel
Bagel
% Bagel instproc flavor {args} {
  $self set flavors $args
}
% Bagel instproc size {s} {
  $self set bites $s
}
% Bagel abagel -flavor Sesame -size 12
abagel
% abagel info vars
flavors bites
% abagel set flavors
Sesame
% abagel set bites
12
```

Once created, objects are manipulated through methods placed on them directly, and methods they inherit from their class object and its superclasses. The former methods are called procs, the latter instprocs. As in Tcl, there is no distinction between system provided methods (such as info and set) and user provided methods (such as flavor and size). Procs take precedence over instprocs, and the order of inheritance for instprocs is discussed under the `superclass` instproc in OTcl Classes.

All methods are called through the object by using a widget-like syntax. The method name is used as the first argument, with the method's arguments as subsequent arguments. The most specific method that is found either on the object or in its inheritance ordering will be invoked. The flavor and size methods are called on bagels as follows.

```
% abagel flavor Sesame Onion
Sesame Onion
% abagel size 10
10
```

```
% abagel set flavors
Sesame Onion
% abagel set bites
10
```

Generic objects may be created with the `Object` class. `Object` is the repository for the behavior common to all objects. It includes methods for defining new methods and instance variables, initializing and destroying objects, querying them, and so forth. The remainder of this reference page describes these methods. Their functionality can be customized for particular classes or objects by using the standard inheritance mechanisms, or changed directly for all objects by rewriting the methods on `Object` in Tcl or C.

Alloc

The `alloc` proc is used to allocate a fresh object that is an instance of class `Object`. It is normally called by the system as part of object creation, but may be called by the user.

The system `create` instproc on `Class` expects all `alloc` procs to take the name of the object to allocate, and a list of arguments. It expects them to allocate the object, install it in the interpreter, and return the list of unprocessed arguments. For the case of the `Object alloc` proc, no additional arguments are processed, and so they are all returned.

To customize object creation, write an `init` instproc, not an `alloc` proc. New `alloc` procs will typically be written in C to allocate structurally different types of object.

```
% Object alloc foo bar baz
bar baz
% foo info class
Object
% foo info procs
% foo info vars
```

Array

The `array` instproc returns information about array instance variables. It mirrors the Tcl array command. See the Tcl array command for options. Array is conceptually defined as follows.

```
Object instproc array {opt ary args} {
  $self instvar $ary
  eval array [list $opt] [list $ary] $args
}
```

Class

The `class` instproc changes the class of an object, where the notion of class is expressed using the names of class objects. The class of an object may be changed at any time, with a run-time type checking system enforcing safety as subsequent methods are executed.

```
% Class Bagel
Bagel
% Bagel instproc what {} { $self info class }
% Class NewBagel
NewBagel
% Bagel abagel
abagel
% abagel info class
Bagel
% abagel what
Bagel
% abagel set foo bar
bar
% abagel info vars
foo
% abagel class NewBagel
% abagel info class
NewBagel
% abagel what
abagel: unable to dispatch method what
% abagel info vars
foo
```

Changing the class of an object does not change the instance variables and procs it contains, only the instprocs accessible through it. This may be customized with the standard inheritance mechanisms.

Destroy

The `destroy` instproc tears down the object, removes it from the interpreter, and releases its memory. Unset traces on instance varaibles are triggered in the process. It takes no arguments, and returns the empty string.

User defined teardown code may be added to objects and classes with the standard inheritance mechanisms.

If the object is also a class, then its instances are destroyed, and classes that depend on it as superclasses have it removed from their superclass list.

There is only one user visible `destroy` method for both objects and classes. The real teardown work is performed below the method level, through deletion callbacks issued by the Tcl interpreter. This ensures that cleanup will be invoked if the command

corresponding to the object is deleted from the interpreter in any manner, including calls to `Tcl_DeleteCommand` and renaming the command to {}.

```
% Class Bagel
Bagel
% Bagel instproc destroy {} {
  puts "zap!"
  $self next
}
% Bagel abagel
abagel
% abagel proc destroy {} {
  puts "poof!"
  $self next
}
% abagel destroy
poof!
zap!
% info commands abagel
```

While cleanup (including user defined teardown) occurs even if the command is renamed to {}, calling the `destroy` method is the preferable way to dispose of an object, since error codes cannot be returned if `Tcl_DeleteCommand` is triggered directly.

Info

The `info` instproc is used to query the object and retrieve information about its current state. It mirrors the Tcl info command, and has the following options.

- `class` returns the class of the object. With an additional argument that is the name of a class, it returns 1 if the object is a direct or indirect instance of that class, and 0 otherwise.
- `procs` returns a list of the names of proc methods defined on the object. An additional argument is taken to be a string match pattern which filters the result list.
- `commands` returns a list of the names of both Tcl and C proc methods defined on the object. An additional argument is taken to be a string match pattern which filters the result list.
- `args` is used to query the argument list of a Tcl proc method. It functions in the same manner as the Tcl info args command.
- `body` is used to query the body of a Tcl proc method. It functions in the same manner as the Tcl info body command.
- `default` is used to query the default value of an argument of a Tcl proc method. It functions in the same manner as the Tcl info default command.
- `vars` returns a list of the names of instance variables defined on the object. An additional argument is taken to be a string match pattern which filters the result list.

In conjunction with other methods such as `array` and `set`, these options can recover most information about the state of an object. As an example, the following proc reverse engineers Tcl procs. This is its output when run on itself.

```
Object proc retrieve {p} {
   set txt [list $self proc $p]
   set al [$self info args $p]
   set dft {}
   for {set i 0} {$i < [llength $al]} {incr i} {
     set av [lindex $al $i]
     if {[$self info default $p $av dft]} then {
       set al [lreplace $al $i $i [list $av $dft]]
     }
   }
   lappend txt $al
   lappend txt [$self info body $p]
   return $txt
}
```

Init

The `init` instproc is used to initialize a freshly allocated object that is a direct or indirect instance of the class `Object`. It is normally called by the system (perhaps from more specialized `init` instprocs) as part of object creation, but may be called by the user.

`init` interprets its arguments as pairs of option keys and option values. Each option key should be the name of a valid method for the object, preceded by a dash. The method should take one argument. For each option key and option value pair, init calls the method on the object, with the option value as its argument. It returns the empty string.

To customize object creation, write an `init` instproc for a class, not an `alloc` proc. If the option key and option value creation syntax is still desired, then call the `Object init` instproc by using `next`. This is discussed in the `create` instproc in OTcl Classes.

```
% Class Bagel
Bagel
% foreach i {1 2 3 4} {
  Bagel instproc $i {v} {puts $v}
}
% Bagel abagel
abagel
% abagel init -1 one -2 two -3 three -4 four!
one
two
three
four!
```

`init` is conceptually equivalent to the following.

```
Object instproc init {args} {
  if {[llength $args]%2 != 0} then {
    error {uneven number of arguments}
  }
  while {$args != {}} {
    set key [lindex $args 0]
    if {[string match {-*} $key]} then {
      set key [string range $key 1 end]
    }
    set val [lindex $args 1]
    if {[catch {$self $key $val} msg]!=0} then {
      set opt [list $self $key $val]
      error "$msg during $opt"
    }
    set args [lrange $args 2 end]
  }
  return {}
}
```

Instvar

The `instvar` instproc is used within the body of a method to map instance variables to local variables. It mirrors the Tcl upvar command (and is implemented in terms of it).

Multiple instance variables may be declared at once. The instance variables may be scalars or arrays (but see below), and need not be previously defined. By default the local alias for an instance variable is the same as the name of the instance variable. Renaming in the style of upvar is specified using a two element lists in the declaration. This departs from Tcl upvar syntax, but allows a simple declaration for the majority of cases, with access to the full functionality of upvar when necessary.

```
% Class Bagel; Bagel abagel
abagel
% abagel set flavor sesame
sesame
% abagel set size {12 bites}
12 bites
% abagel proc taste {} {
  $self instvar flavor
  return $flavor
}
% abagel taste
sesame
% abagel proc query {} {
  $self instvar size {flavor f}
  return "$f, $size"
}
% abagel query
sesame, 12 bites
```

Note that the renaming syntax is required to access individual array elements directly. This is because Tcl's upvar does not allow a remote array element to be locally accessed as an array element. Instead, it is often easier to map the whole array for access.

Next

The `next` instproc is used within the body of a method to call the next-most shadowed method. It is used to combine inherited methods without depending on explicitly knowing their location. For example, it is typically used as part of `init` instprocs for classes to form an aggregate initialization method. `next` is analogous to call-next-method in CLOS. See the `superclass` instproc in OTcl Classes for a discussion of the order of inheritance.

`next` searches for an instproc method with the same name as the current method. It begins its search after the position in the precedence ordering where the current method was found, if the current method is an instproc, or from the beginning of the precedence ordering of the object's class, if the current method is a proc. The position information between calls to `next` is recovered from the `class` variable. If no next method is found, then an empty string is returned without error. Otherwise, the next method is called with the arguments that were passed to `next`.

Proc

The `proc` instproc is used to install proc methods on an object, for sole use by that object. Use `proc` to customize individual objects beyond the functionality provided by via their class, not for inheritance. With particular argument forms, `proc` can also remove proc methods from an object, or specify an autoload script for demand loading of the proc method.

The arguments and body of a proc method are of the same form as a Tcl procedure, with two exceptions. If both args and body are empty, then an existing proc method with the specified name is removed from the object. If args is `{auto}`, then the body is interpreted as an autoload script as described below.

Within the body of the proc and instproc methods, three special variables are defined. These variables are for reading only. Instance variables may be accessed as local variables by using the `instvar` instproc.

- `self` is bound to the name of the object on whose behalf the method is executing. It may be used to invoke further methods. It is the equivalent of this in C++.
- `class` is bound to the name of the class object on which the method that is executing is defined, if the method is an instproc, and the empty string if the method is a proc. It does not contain the class of the object, which may be retrieved with the `info` instproc.
- `proc` is bound to the name of the proc or instproc method that is executing.

```
% Class Bagel; Bagel abagel
abagel
% abagel info procs
% abagel proc flavor {f} {
  $self instvar flavor
  set flavor $f
  return "called $self $proc $f"
}
% abagel info procs
flavor
% abagel flavor sesame
called abagel flavor sesame
```

Proc and instproc methods may also be declared to autoload. This function is usually accessed through the higher level demand loading scheme described in OTcl Autoloading.

If the argument list is `{auto}`, then the body is taken to be a script for demand loading of the method. When the method is invoked, the script will be executed (and should cause the real method to be loaded) and then the method will be restarted. While the stub is waiting to load the method body, the method is recognized as a `proc` by the `info` method, but cannot be queried for its body or arguments.

```
% set tmp [open "tmp" w]
file3
% puts $tmp {abagel proc bagel {} { return "bagel" }}
% close $tmp
% Class Bagel; Bagel abagel
abagel
% abagel proc bagel auto {
    puts -nonewline "loading... "
    source tmp
  }
% abagel bagel
loading... bagel
% abagel bagel
bagel
```

Set

The `set` instproc is used to place instance variables on an object as well as to access them. It mirrors the Tcl set command (and is implemented in terms of it). It returns the value of the instance variable. Instance variables may be scalar or array variables. They are stored in separate slots than methods, and so are distinct from methods with the same name.

```
% Class Bagel; Bagel abagel
abagel
% abagel info vars
% abagel set avar aval
aval
% abagel set avar
```

```
aval
% abagel set foo(bar) baz
baz
% abagel set foo(bar)
baz
% abagel info vars
foo avar
```

Unknown

The `unknown` method, if defined for an object, is invoked by the system when no matching method can be found for regular dispatch. By default, it is not defined for `Object`, but exists as a hook for user defined handlers, such as abbreviations, load monitoring, error reporting, etc. An `unknown` instproc that implements implicit creation is defined for `Class`; see its reference page.

Like Tcl's `unknown` proc, the `unknown` method receives as its arguments the name of the method that could not be invoked, along with that method's arguments. The result it returns is returned as the overall result of the call.

As an example, the following `unknown` instproc implements abbreviations and verbose error messsages.

```
% Object instproc unknown {m args} {
  foreach i [$self info commands] {
    lappend meth($i) {}
  }
  set cl [$self info class]
  foreach i [concat $cl [$cl info heritage]] {
    foreach j [$i info instcommands] {
     lappend meth($j) {}
    }
  }
  set abbrev [array names meth "$m*"]
  switch -exact [llength $abbrev] {
     0 { error "$self: invalid method \"$m\": [lsort [array names
meth]]" }
     1 { eval [list $self] $abbrev $args }
     default { error "$self: ambiguous method \"$m\": [lsort $abbrev]" }
  }
}
% Object obj
obj
% obj f
obj: invalid method "f": array class destroy info init instvar next
proc set unknown unset
% obj i
obj: ambiguous method "i": info init instvar
% obj d
```

Unset

The `unset` instproc is used to remove instance variables from an object. It mirrors the Tcl unset command (and is implemented in terms of it).

```
% Class Bagel; Bagel abagel
abagel
% abagel set foo bar
bar
% abagel info vars
foo
% abagel unset foo
% abagel info vars
```

OTcl Classes

Overview

This reference page describes the functionality provided for all classes by methods on the `Class` class. See the tutorial for an introduction to programming in OTcl, and the C API page for details of manipulating classes from C.

Classes in OTcl are a special kind of object used mainly for inheritance. By convention, they are named with mixed case to distinguish them from regular objects. They inherit all the abilities of regular objects from `Object`, and add more. The inherited behavior includes method lookup, dispatch, and combination, and initialization syntax. See Objects in OTcl to understand these abilities of regular objects.

Classes are created through meta-class objects, either implicitly with a widget command syntax, or explicitly by calling a creation method. Generic classes may be created with the `Class` class. By default, they will inherit from `Object`. The classes Bagel, and SuperBagel, which inherits from Bagel, may be defined as follows. See the `create` instproc for a description of the overall creation process and how to customize it.

```
% Class Bagel
Bagel
% Bagel info class
Class
% Bagel info superclass
Object
% Class SuperBagel -superclass Bagel
SuperBagel
% SuperBagel info class
Class
% SuperBagel info superclass
Bagel
```

`Class` is the repository for the behavior common to all classes. It includes methods for defining new methods for use by instances, initializing and destroying classes, specifying their superclasses, querying them, and so forth. The remainder of this reference page describes these methods. Their functionality can be customized for particular meta-classes or classes by using the standard inheritance mechanisms, or changed directly for all classes by rewriting the methods on `Class` in Tcl or C.

Alloc

The `alloc` proc is used to allocate a fresh class object that is an instance of class `Class` and has superclass `Object`. It is normally called by the system as part of class creation, but may be called by the user.

The system `create` instproc on `Class` expects all `alloc` procs to take the name of the object to allocate, and a list of arguments. It expects them to allocate the object, install it in the interpreter, and return the list of unprocessed arguments. For the case of the `Class alloc` proc, no additional arguments are processed, and so they are all returned.

To customize class creation, write an `init` instproc, not an `alloc` proc. New `alloc` procs will typically be written in C to allocate structurally different types of object.

```
% Class alloc foo bar baz
bar baz
% foo info class
Class
% foo info procs
% foo info vars
```

Create

The `create` instproc provides a mechanism for classes to create other classes and objects. It is invoked by the default `unknown` instproc if no matching method name can be found, and so may be ellided to yield the familiar widget-like creation syntax.

`create` takes the name of a class or object to create plus extra initialization argument pairs, and returns the name of the object or class created. It effectively calls an `alloc` proc to allocate the object, then dispatches the `init` method with the initialization arguments to initialize the object. Recall that the base `init` method on `Object` interprets these arguments as option key and option value pairs, evaluating each pair in turn. The following three sequences are essentially equivalent (except in terms of return value).

```
% Class create Bagel
Bagel

% Class Bagel
Bagel

% Class alloc Bagel
% Bagel init
```

The `alloc` proc that is called by create is determined by the class object on which it is called. `create` first looks for an `alloc` proc on this class object, then on its superclasses according to the precedence ordering. The result is that calling `create` on `Class` (or a class specialized from it) will create a new class, whereas calling `create` on `Object` (or a class specialized from it) will create a new object.

Classes may customize the initialization of their instances by defining an `init` instproc. If the option key and option value creation syntax is still desired, this `init` instproc should combine its behavior with the `init` instproc on `Object` by using the `next` instproc, as shown below. For example, the class Bagel may require an instance variable called

bites to be initialized to a default value of 12, in addition to regular option key and option value initialization. This is accomplished as follows. (The use of eval is simply to flatten the list of arguments contained in args.) Similarly, an `init` instproc on `Class` may be used to customize the initialization of all classes.

```
% Class Bagel
Bagel
% Bagel instproc init {args} {
  $self set bites 12
  eval $self next $args
}
% Bagel instproc flavor {f} {
  $self set flavor $f
}
% Bagel abagel -flavor sesame
abagel
% abagel set bites
12
% abagel set flavor
sesame
```

Alternatively, the standard inheritance mechanisms may be used to provide some or all classes with their own `create` proc, allowing complete control over the creation process. For reference, the default `create` instproc on `Class` is conceptually defined as follows.

```
Class instproc create {obj args} {
  set h [$self info heritage]
  foreach i [concat $self $h] {
    if {[$i info commands alloc] != {}} then {
      set args [eval [list $i] alloc [list $obj] $args]
      $obj class $self
      eval [list $obj] init $args
      return $obj
    }
  }
  error {No reachable alloc}
}
```

Info

The `info` instproc is used to query the class and retrieve information about its current state. It mirrors the Tcl info command, and has the following options in addition to those of the `Object info` instproc.

- `superclass` returns the superclass list of the object. With an additional argument that is the name of a class, it returns 1 if that class is a direct or indirect superclass of the class, and 0 otherwise.

- `subclass` returns the subclass list of the object. With an additional argument that is the name of a class, it returns 1 if that class is a direct or indirect subclass of the class, and 0 otherwise.
- `heritage` returns the inheritance precedence list (as described for the `superclass` instproc). An additional argument is taken to be a string match pattern which filters the result list.
- `instances` returns a list of the instance objects of the class. An additional argument is taken to be a string match pattern which filters the result list.
- `instprocs` returns a list of the names of instproc methods defined on the class. An additional argument is taken to be a string match pattern which filters the result list.
- `instcommands` returns a list of the names of both Tcl and C instproc methods defined on the class. An additional argument is taken to be a string match pattern which filters the result list.
- `instargs` is used to query the argument list of a Tcl instproc method. It functions in the same manner as the Tcl info args command.
- `instbody` is used to query the body of a Tcl instproc method. It functions in the same manner as the Tcl info body command.
- `instdefault` is used to query the default value of an argument of a Tcl instproc method. It functions in the same manner as the Tcl info default command.

These options can recover most information about the state of a class. As an example, the following instproc returns a list of all direct and indirect instances of a class.

```
Class instproc instances {} {
  set il {}
  foreach i [Class info instances] {
    if {[$self info subclass $i]} then {
      eval lappend il [$i info instances]
    }
  }
  return $il
}
```

Instproc

The `instproc` instproc is used to install instproc methods on a class, for use by that class's direct and indirect instances. Use `instproc` to share and inherit behaviors. With particular argument forms, `instproc` can also remove instproc methods from a class, or specify an autoload script for demand loading of the instproc method.

The arguments and body of an instproc method are of the same form as a Tcl procedure, with two exceptions. If both args and body are empty, then an existing instproc method with the specified name is removed from the class. If args is `{auto}`, then the body is interpreted as an autoload script in the same manner as described under the `proc` instproc in OTcl Objects. See OTcl Autoloading for a higher level demand loading scheme.

The following example demonstrates defining instprocs, using them on behalf of and object, and combining their functionality with next.

```
% Class Bagel
Bagel
% Class SuperBagel -superclass Bagel
SuperBagel
% Bagel instproc taste {} {
  puts yum!
}
% SuperBagel instproc taste {} {
  $self next
  puts YUM!
}
% SuperBagel abagel
abagel
% abagel taste
yum!
YUM!
```

The environment in effect when an instproc is being executed is the same as when a proc is being executed, and is described under the `proc` instproc in OTcl Objects. The special variable `class` may be used for a variety of tasks, such as to access shared variables stored on the class object. For example, the default size of a bagel in bites may be stored on the Bagel class to be accessed during the `init` instproc as follows.

```
% Class Bagel
Bagel
% Bagel set bites 12
12
% Bagel instproc init {args} {
  $class instvar bites
  $self set size $bites
  eval $self next $args
}
% Bagel abagel
abagel
% abagel set size
12
% Bagel set bites 7
7
% Bagel abagel
abagel
% abagel set size
7
```

Superclass

The `superclass` instproc is used to change the superclasses from which a class directly inherits behavior. It takes one argument that is a list of superclasses and returns the empty

string. The order of the superclass list determines the order of inheritance. Multiple inheritance is supported.

The superclasses must be in precedence order (from most specialized to least specialized) if they are related, and the resulting superclass relation must be cycle-free. An error is returned and the superclass graph is unchanged if either of these conditions are unmet.

The linear precedence order in which superclasses are searched for instprocs is constrained according to each local superclass list. It is guaranteed that if A inherits from B and C, then A will behave like a B before it behaves like a C, and so forth for B and C and their superclasses. The algorithm used to generate this ordering is an unspecified CLOS-like topological sort of the inheritance graph. (This is all you need to know. Multiple inheritance is best thought of in terms of local orderings. If you are relying on subtleties of the global ordering, then you are asking for trouble.)

The `heritage` option of the `info` instproc may be used to discover the precedence order, and hence the path that the `next` instproc will use when instructed to combine methods.

Unknown

The `unknown` instproc for classes is used to implement implicit creation of objects. It is invoked when no matching method is found, and interprets the method name as the name of an object to be created with `create`, thus allowing a widget-like creation syntax.

See the entry for `unknown` in OTcl Objects for a general description of the unknown method mechanism.

The `unknown` instproc on `Class` is conceptually defined as follows. It you do not want implicit creation, then redefine or remove the default method with the `instproc` method.

```
Class instproc unknown {m args} {
  if {$m == {create}} then {
    error "$self: unable to dispatch $m"
  }
  eval [list $self] create [list $m] $args
}
```

OTcl Autoloading

Overview

This reference page describes how arrange for OTcl classes and their methods to be demand loaded. The mechanism extends the existing Tcl autoloading scheme (with which you should be familiar) by using an `otcl_mkindex` procedure to add entries to a `tclIndex` file. It allows you to distribute classes across files without additional concern for inheritance dependencies, as well as distribute a class's methods across files.

Extending a tclIndex

To add OTcl load entries to a `tclIndex`, use the `otcl_mkindex` procedure, defined during OTcl initialization. For example, to update the `tclIndex` in the current directory for the demand loading of classes, issue this shell command:

```
echo "otcl_mkindex Class . *.tcl" | otclsh
```

`otcl_mkindex` takes the same directory and filename pattern arguments as `auto_mkindex`, but also takes a list of classes as its first argument. This list, the creator list, describes the kind of objects that are candidates for demand loading. Usually this will be `Class`, as above, to cause index entries to be generated for classes. But it may also specify other classes if you are working with meta-classes or need loading for regular objects. `otcl_mkindex` appends the `tclIndex` file directly, and returns a string describing the number of object and method index entries it generated.

Like `auto_mkindex`, `otcl_mkindex` will only find obvious candidates for demand loading. Both explicit creation, via the `create` method, and implicit creation, via a method name that is not a known method of `Class`, are considered in the search. Objects (but not methods) must be created at the hard left margin, and object and method names may not begin with "$".

How Loading Works

Autoloading of class and object commands is triggered via the regular Tcl `unknown` mechanism, so that invoking a missing class or object command will cause it to be loaded. In addition, OTcl demand loads in two other ways.

First, unknown classes referenced as superclasses will be demand loaded. This is triggered internally by the OTcl system. It means that you can distribute classes across files without concern for sourcing the files in inheritance order. It does not mean, however, that you can make forward references to classes within a file, or mutually recursive forward references across files.

Second, undefined methods may be demand loaded when they are invoked. This is arranged for autoloaded classes by the OTcl object loader, `otcl_load`. It means that you can distribute the definition of a class and its methods over more than one file.

Method level loading works as follows. When a class or object is being autoloaded by `otcl_load`, only those methods defined in the main class file that is sourced will be fully loaded. Other methods that are known to exist because of `auto_index` entries are then filled with load stubs by using the `auto` option of the `proc` and `instproc` methods. This guarantees that if they are invoked, they will be loaded. Note that these method load stubs must be installed, rather than relying on an unknown-style load scheme, to cater for the shadowing of methods. Also, you can control your own load policy by rewriting `otcl_load`.

OTcl C API

Overview

This reference page describes the C application programmer interface (API) for manipulating OTcl classes and objects. See otclAppInit.c for an example of a `Timer` class written in C; `Timer` is included in the shells if the symbol `TESTCAPI` is defined.

OTcl's C API is designed to complement the OTcl language in much the same way that Tcl allows commands written in C to be added to an interpreter. It is a minimal interface, suitable for migrating methods to C for performance, or for manipulating complex (ie. non-string) data structures. It is not a general package for binding C++ classes and methods to Tcl commands.

See the tutorial for an introduction to Tcl-level programming in OTcl.

Working with the C API

To access the C API, include `otcl.h`. This header defines the externally visible interfaces, including the data structures required to use them.

Object and Class Structures

Objects and classes are always manipulated through pointers to opaque structures:

```
struct OTclObject;
struct OTclClass;
```

Actually, you can cast a class pointer to an object pointer (since all classes are objects too) with no ill-effects, but you shouldn't need to.

Two utility functions convert string names to object and class pointers:

```
struct OTclObject*
OTclGetObject(Tcl_Interp* in, char* name);

struct OTclClass*
OTclGetClass(Tcl_Interp* in, char* name);
```

These functions are useful for getting handles to `Object` and `Class` (the system provided classes) for use in creating new classes, etc.

Two utility functions convert clientdata to object and class pointers:

```
struct OTclObject*
OTclAsObject(Tcl_Interp* in, ClientData cd);

struct OTclClass*
OTclAsClass(Tcl_Interp* in, ClientData cd);
```

These functions are useful within method definitions when you have installed the method to pass the current object via the clientdata; see the section on Methods below. They perform a safe cast.

Initialization

OTcl is initialized for an interpreter with a standard module initialization routine, called from an AppInit or through dynamic loading.

```
int
Otcl_Init(Tcl_Interp* in);
```

Calls through the C API must be arranged to occur after OTcl initialization.

Objects and Classes

Objects and classes can be created and destroyed from C as well as Tcl without distinction. That is, classes created in C can be destroyed from Tcl, and vice versa.

Creation

Objects and classes are created in an interpreter through class pointers. Use a pointer to `Class` to create a generic class, and a pointer to `Object` to create a generic object.

```
struct OTclObject*
OTclCreateObject(Tcl_Interp* in, char* name, struct OTclClass* cl);

struct OTclClass*
OTclCreateClass(Tcl_Interp* in, char* name, struct OTclClass* cl);
```

Both calls are conceptually equivalent to `"cl create name"` in Tcl, but return either a pointer value or `NULL` to indicate failure.

Deletion

Object and classes are deleted from an interpreter through their pointers.

```
int
OTclDeleteObject(Tcl_Interp* in, struct OTclObject* obj);
```

```
int
OTclDeleteClass(Tcl_Interp* in, struct OTclClass* cl);
```

Both calls are conceptually equivalent to "`obj destroy`" or "`cl destroy`", and return a Tcl call code.

Methods

Methods can be added and combined from C as well as from Tcl without distinction. For example, C methods can be called from Tcl transparently, and C methods can combine with Tcl methods automatically.

Interface Conventions

In terms of interface, methods are analogous to Tcl commands, with two important differences.

1. The argc/argv array is passed in "expanded form", having three extra arguments. argv[0] contains the name of the object, just as Tcl's first argument contains the name of the invoking command. The next three arguments contain values for extra method context variables: `self`, `class`, and `proc`. The remaining arguments (argv[4] and higher) contain the arguments passed to the method.
2. The clientData may be used to obtain a pointer to the object on behalf of which the method is being invoked. If the method was created with a clientData of `NULL`, then the dispatcher fills the clientData with an object pointer. Otherwise, the dispatcher passes the specified clientData. To convert the clientData to an object or class pointer, you can use the typed casting functions `OTclAsObject` and `OTclAsClass`.

Adding Methods

Two functions add methods to objects and classes, serving as the C equivalent of the `proc` and `instproc` methods. The types of the last three arguments are the same as for Tcl commands.

```
void
OTclAddPMethod(struct OTclObject* obj, char* nm, Tcl_CmdProc* proc,
               ClientData cd, Tcl_CmdDeleteProc* dp);

void
OTclAddIMethod(struct OTclClass* cl, char* nm, Tcl_CmdProc* proc,
               ClientData cd, Tcl_CmdDeleteProc* dp);
```

Removing Methods

Two functions remove methods from objects and classes. If a deleteProc callback was registered to clean up the method, then it is passed the original non-NULL clientdata. If

the original clientdata was NULL, however, then a pointer to the object or class from which the method is being removed is passed instead.

```
int
OTclRemovePMethod(struct OTclObject* obj, char* nm);

int
OTclRemoveIMethod(struct OTclClass* cl, char* nm);
```

Combining Methods

An executing methods can be automatically combined with the next-most specific method with the following function. It is equivalent to `"obj next ..args.."`. argc/argv is passed in expanded form, and should carry the context of the currently executing method. It returns a Tcl return code.

```
int
OTclNextMethod(struct OTclObject* obj, Tcl_Interp* in, int argc,
char*argv[]);
```

Instance Variables

Tcl accessible instance variables (stored as strings) can be manipulated from C. In addition, objects can store a handle to private auxilliary data.

*InstVar

OTclSetInstVar, OTclGetInstVar, and OTclUnsetInstVar mimic Tcl_SetVar, Tcl_GetVar, and Tcl_UnsetVar for instance variables. The return values and codes for parameters such as `flgs` match Tcl conventions.

```
char*
OTclSetInstVar(struct OTclObject* obj, Tcl_Interp* in,
               char* name, char* value, int flgs);
char*
OTclGetInstVar(struct OTclObject* obj, Tcl_Interp* in,
               char* name, int flgs);

int
OTclUnsetInstVar(struct OTclObject* obj, Tcl_Interp* in,
                 char* name, int flgs);
```

Auxilliary Data

OTclSetObjectData, OTclGetObjectData and OTclUnsetObjectData manipulate private object clientdata, such as a pointer to an auxilliary data region. ObjectData is a per object and per class resource to allow for inheritance. Typically, it is manipulated on behalf of

the invoking object in each class method by using the directly associated class. In this manner specializations of a class may each store their own ObjectData.

```
int
OTclGetObjectData(struct OTclObject* obj,struct OTclClass* cl,
                  ClientData* data);

void
OTclSetObjectData(struct OTclObject* obj, struct OTclClass* cl,
                  ClientData data);

int
OTclUnsetObjectData(struct OTclObject* obj, struct OTclClass* cl);
```

Get fills the `data` value passed by reference, and returns `0` or `1` depending on whether the ObjectData existed. If it didn't, then data is filled with `NULL`. Set overwrites existing ObjectData without error. Unset returns `0` or `1` depending on whether the ObjectData existed.