NAME

oo::define, **oo::objdefine** -
define and configure classes and objects

SYNOPSIS

```
package require TclOO

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

DESCRIPTION

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

CONFIGURING CLASSES

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

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

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

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

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

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

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

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

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

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

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

self *subcommand arg ...*

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

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

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

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

CONFIGURING OBJECTS

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

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

deletemethod *name ?name ...*
: This deletes each of the methods called *name* from an object. The methods must have previously existed in that object. Does not affect the classes that the object is an instance of.

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

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

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

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

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

renamemethod *fromName toName*
: This renames the method called *fromName* in an object to *toName*. The method must have previously existed in the object, and *toName* must not previously refer to a method in that object. Does not affect the classes that the object is an instance of. Does not change the export status of the method; if it was exported before, it will be afterwards.

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

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

SLOTTED DEFINITIONS

Some of the configurable definitions of a class or object are *slotted definitions*. This means that the configuration is implemented by a slot object, that is an instance of the class **oo::Slot**, which manages a list of values (class names, variable names, etc.) that comprises the contents of the slot. The class defines three operations (as methods) that may be done on the slot:

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

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

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

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

SLOT IMPLEMENTATION

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

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

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

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

EXAMPLES

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

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

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

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

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

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

SEE ALSO

next(n), oo::class(n), oo::object(n)

KEYWORDS

class, definition, method, object, slot