nx::Object(3) 2.0 Object ""

NAME

nx::Object - API reference of the base class in the NX object system

TABLE OF CONTENTS

SYNOPSIS

nx::Object create *obj* ?**-object-mixins** *mixinSpec*? ?**-class** *newClassName*? ?**-object-filters** *filterSpec*? ?*initBlock*?
nx::Object new ?**-object-mixins** *mixinSpec*? ?**-class** *newClassName*? ?**-object-filters** *filterSpec*? ?*initBlock*?
obj ?**public** | **private** | **protected**? **object alias** *methodName* ?**-returns** *valueChecker*? ?**-frame object** | **method**? *cmdName*
obj **cget** *configurationOption*
obj **configure** ?*configurationOption value* ...?
obj **contains** ?-withnew *trueFalse*? ?-object *objectName*? ?-class *className*? *cmds*
obj **copy** *newObjectName*
obj **delete object** *feature arg*
obj **destroy**
obj **eval** *arg* ?*arg* ...?
obj **object filters** *submethod* ?*arg* ...?
obj ?**public** | **protected** | **private**? **object forward** *methodName* ?**-prefix** *prefixName*? ?**-frame object**? ?**-returns** *valueChecker*? ?**-verbose**? ?*target*? ?*arg* ...?
obj **info children** ?**-type** *className*? ?*pattern*?
obj **info class**
obj **info has** ?**mixin** | **namespace** | **type**? ?*arg* ...?
obj **info lookup** *submethod* ?*arg* ...?
obj **info name**
obj **info info** ?**-asList**?
obj **info object filters** ?**-guards**? ?*pattern*?
obj **info object method** *option methodName*
obj **info object methods** ?**-callprotection** *level*? ?**-type** *methodType*? ?**-path**? ?*namePattern*?
obj **info object mixins** ?**-guards**? ?*pattern*?
obj **info object slots** ?**-type** *className*? ?*pattern*?
obj **info object variables** ?*pattern*?
obj **info parent**
obj **info precedence** ?**-intrinsic**? ?*pattern*?
obj **info variable** *option handle*
obj **info vars** ?*pattern*?

obj ?**public** | **protected** | **private**? **object method** *name parameters* ?**-checkalways**? ?**-returns** *valueChecker*? *body*
obj **move** *newObjectName*
obj **object mixins** *submethod* ?*arg* ...?
obj **object property** ?**-accessor public** | **protected** | **private**? ?**-configurable** *trueFalse*? ?**-incremental**? ?**-class** *className*? ?**-nocomplain**? *spec* ?*initBlock*?
obj **require namespace**
obj **require** ?**public** | **protected** | **private**? **object method** *methodName*
obj **unknown** *unknownMethodName* ?*arg* ...?
obj **object variable** ?**-accessor public** | **protected** | **private**? ?**-incremental**? ?**-class** *className*? ?**-configurable** *trueFalse*? ?**-initblock** *script*? ?**-nocomplain**? *spec* ?*defaultValue*?

DESCRIPTION

`nx::Object` is the base class of the NX object system. All objects defined in NX are (direct or indirect) instances of this base class. The methods provided by the `nx::Object` base class are available to all objects and to all classes defined in NX.

```
 +---------+
 | ::nx::* |
 +---------+--------------------------------------Y
 |                                                |
 |  +---------+     instance of     +----------+  |
 |  |         |<....................|          |  |
 |  |  Class  |                     |  Object  |  |
 |  |         |....................>|          |  |
 |  +----+----+     subclass of     +-----+----+  |
 |       ^                           ^    ^       |
instance.|...........................|....|......./
      of |                           |    |
   +-----+-----+    subclass of      |    | instance
   |           |.....................|    | of
   |   /cls/   |    (by default)          |
   |           |                          |
   +-----------+                          |
         ^                                |
instance |.............(xor)..............|
      of |         +-----------+          |
         |.........|           |..........|
                   |   /obj/   |
                   |           |
                   +-----------+
```

NX allows for creating and for using objects (e.g. *obj*) which are instantiated from the base class `nx::Object` directly. Typical use cases are singletons and anonymous, inline objects. In such use cases, NX does not require creating an intermediate application class (e.g. *cls*), which specializes the base class `nx::Object` by default, beforehand.

Objects (e.g. *obj*) which are creating by instantiating a previously defined application class (e.g. *cls*) are indirect instances of `nx::Object`.

Direct instances of `nx::Object` can be created as follows:

`nx::Object` `create` *`obj`* ? `-object-mixins` *`mixinSpec`* ? ? `-class` *`newClassName`* ? ? `-object-filters` *`filterSpec`* ? ? *`initBlock`* ?

> To create a direct instance of `nx::Object` having an explicit name *`obj`*, use `create` on `nx::Object`. Note that `create` is defined by `nx::Class` and is available to `nx::Object` being an instance of `nx::Class`. This way, singleton objects can be created, for example.

`nx::Object` `new` ? `-object-mixins` *`mixinSpec`* ? ? `-class` *`newClassName`* ? ? `-object-filters` *`filterSpec`* ? ? *`initBlock`* ?

> To create a direct instance of `nx::Object` having an automatically assigned, implict object name, use `new` on `nx::Object`. Note that `new` is defined by `nx::Class` and is available to `nx::Object` being an instance of `nx::Class`. Using `new` allows for creating anonymous, inline objects, for example.

The configuration options for direct and indirect instances of `nx::Object`, which can be passed when calling **create** and **new**, are documented in the subsequent section.

CONFIGURATION OPTIONS FOR INSTANCES OF NX::OBJECT

Configuration options can be used for configuring objects during their creation by passing the options as non-positional arguments into calls of **new** and **create** (see `nx::Class`). An existing object can be queried for its current configuration using **cget** and it can be re-configured using **configure**. Legal configuration options are:

`-class` ? *`className`* ?

> Retrieves the current class of the object or sets the object's class to *`className`*, if provided.

`-object-filters` ? *`filterMethods`* ?

> Retrieves the list of currently active per-object filter methods or sets a list of per-object filter methods, if *`filterMethods`* is provided.

`-object-mixins` ? *`mixinSpecs`* ?

> If *`mixinSpecs`* is not specified, retrieves the list of currently active per-object mixin specifications. If *`mixinSpecs`* is specified, sets a list of per-object mixin specifications to become active. mixin classes are returned or set in terms of a list of mixin specifications.

METHODS FOR INSTANCES OF NX::OBJECT

alias

> *`obj`* ? **`public`** | **`private`** | **`protected`** ? **`object alias`** *`methodName`* ? `-returns` *`valueChecker`* ? ? `-frame` `object` | `method` ? *`cmdName`*
>
> > Define an alias method for the given object. The resulting method registers a pre-existing Tcl command *`cmdName`* under the (alias) name *`methodName`* with the object. If *`cmdName`* refers to another **`method`**, the corresponding argument should be a valid method handle. If a Tcl command (e.g., a `proc`), the argument should be a fully qualified Tcl command name. If aliasing a subcommand (e.g., `array exists`) of a Tcl namespace ensemble (e.g., `array`), *`cmdName`* must hold the fully qualified subcommand name (and not the ensemble name of the subcommand).

As for a regular `object method`, `-returns` allows for setting a value checker on the values returned by the aliased command *`cmdName`*.

When creating an alias method for a *C-implemented* Tcl command (i.e., command defined using the Tcl/NX C-API), `-frame` sets the scope for variable references used in the aliased command. If the provided value is `object`, then variable references will be resolved in the context of the called object, i.e., the object upon which the alias method is invoked, as if they were object variables. There is no need for using the colon-prefix notation for identifying object variables. If the value is `method`, then the aliased command will be executed as a regular method call. The command is aware of its called-object context; i.e., it can resolve **`::nx::self`**. In addition, the alias method has access to the method-call context (e.g., **`nx::next`**). If `-frame` is omitted, and by default, the variable references will resolve in the context of the caller of the alias method.

cget

`obj` **`cget`** *`configurationOption`*

The method is used to obtain the current value of *`configurationOption`* for *`obj`*. The configuration options available for querying through **`cget`** are determined by the configurable properties defined by the class hierarchy of *`obj`*. The queriable configuration options for *`obj`* can be obtained by calling **`info configure`**. The *`configurationOption`* can be set and modified using **`configure`**.

```
% nx::Object create obj
::obj
% ::obj info configure
?-object-mixins /mixinreg .../? ?-class /class/? ?-object-filters /filterreg .../?
% ::obj cget -class
::nx::Object
```

configure

`obj` **`configure`** **?***`configurationOption`* *`value`* **...?**

This method sets configuration options on an object. The configuration options available for setting on *`obj`* are determined by the configurable properties defined by the class hierarchy of *`obj`*. The settable configuration options for *`obj`* can be obtained by calling **`info configure`**. Furthermore, **`configure`** is also called during object construction. Under object construction, it receives the arguments passed into calls of **`create`** and **`new`**. Options set using **`configure`** can be retrieved using **`cget`**.

```
% nx::Class create Foo {:property x}
::Foo
% Foo create f1 -x 101
::f1
% f1 cget -x
101
% f1 configure -x 200
% f1 cget -x
200
```

contains

`obj` **`contains`** **?-withnew** *`trueFalse`* **? ?-object** *`objectName`* **? ?-class** *`className`* **?** *`cmds`*

This method acts as a builder for nested object structures. Object and class construction statements passed to this method as its last argument *`cmds`* are evaluated in a way so that the receiver object *`obj`* becomes the parent of the newly constructed objects and classes. This is realized by setting explicitly the namespace for constructing relatively named objects. Fully qualified object names in *`cmds`* evade the nesting.

`-withnew` requests the automatic rescoping of objects created using **`new`** so that they become nested into the receiver object *`obj`*, rather than being created in the default namespace for autonamed objects (i.e., ::nsf). If turned off, autonamed objects do not become children of *`obj`*.

The parent object *`objectName`* to be used instead of *`obj`* can be specified using `-object`. If this explicitly set parent object does not exist prior to calling **`contains`**, it will be created on the fly as a direct instance of **`nx::Object`**. Alternatively, using `-class`, a class *`className`* other than **`nx::Object`** for the on-the-fly creation of *`objectName`* can be provided.

```
% nx::Class create Window {
  :contains {
    #
    # Become children of Window, implicitly
    #
    nx::Class create Header; # Window::Header
    nx::Object create Panel; # Window::Panel
  }
  #
  # Explicitly declared a child of Window using [self]
  #
  nx::Class create [self]::Slider; # Window::Slider
  #
  # Fully-qualified objects do not become nested
  #
  nx::Class create ::Door; # ::Door
}
::Window
% ::Window info children
::Window::Panel ::Window::Header ::Window::Slider
```

copy

`obj` **`copy`** *`newObjectName`*

Creates a full and deep copy of a source object *`obj`*. The object's copy *`newObjectName`* features all structural and behavioral properties of the source object, including object variables, per-object methods, nested objects, slot objects, namespaces, filters, mixins, and traces.

delete

`obj` **`delete object`** *`feature`* *`arg`*

This method serves as the equivalent to Tcl's **`rename`** for removing structural (properties, variables) and behavioral features (methods) of the object:

obj **delete object property** *propertyName*

obj **delete object variable** *variableName*

obj **delete object method** *methodName*

Removes a property *propertyName*, variable *variableName*, and method *methodName*, respectively, previously defined for the scope of the object.

delete object method can be equally used for removing regular methods (see **object method**), an alias method (see **object alias**), and a forwarder method (see **object forward**).

destroy

obj **destroy**

This method allows for explicitly destructing an object *obj*, potentially prior to *obj* being destroyed by the object system (e.g. during the shutdown of the object system upon calling **exit**):

```
[nx::Object new] destroy
```

By providing a custom implementation of **destroy**, the destruction procedure of *obj* can be customized. Typically, once the application-specific destruction logic has completed, a custom **destroy** will trigger the actual, physical object destruction via **next**.

```
% [nx::Object create obj {
  :public method destroy {} {
    puts "destroying [self]"
    next; # physical destruction
  }
}] destroy
destroying ::obj
```

A customized object-desctruction scheme can be made shared between the instances of a class, by defining the custom **destroy** for an application class:

```
% nx::Class create Foo {
    :method destroy {} {
      puts "destroying [self]"
      next; # physical destruction
    }
}
::Foo
% Foo create f1
::f1
% f1 destroy
destroying ::f1
```

Physical destruction is performed by clearing the in-memory object storage of *obj*. This is achieved by passing *obj* into a call to **dealloc** provided by **nx::Class**. A near, scripted equivalent to the C-implemented **destroy** provided by **nx::Object** would look as follows:

```
% Object method destroy {} {
  [:info class] dealloc [self]
}
```

Note, however, that `destroy` is protected against application-level redefinition. Trying to evaluate the above script snippet yields:

```
refuse to overwrite protected method 'destroy'; derive e.g. a sub-class!
```

A custom `destroy` must be provided as a refinement in a subclass of `nx::Object` or in a mixin class.

eval

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

Evaluates a special Tcl script for the scope of *`obj`* in the style of Tcl's `eval`. There are, however, notable differences to the standard `eval`: In this script, the colon-prefix notation is available to dispatch to methods and to access variables of *`obj`*. Script-local variables, which are thrown away once the evaluation of the script has completed, can be defined to store intermediate results.

```
% nx::Object create obj {
  :object property {bar 1}
  :public object method foo {x} { return $x }
}
::obj
% ::obj eval {
  set y [:foo ${:bar}]
}
1
```

filters

`obj` `object` **`filters`** *`submethod`* **?** *`arg`* **...?**

Accesses and modifies the list of methods which are registered as filters with *`obj`* using a specific setter or getter *`submethod`*:

`obj` `object` **`filters add`** *`spec`* **?** *`index`* **?**

Inserts a single filter into the current list of filters of *`obj`*. Using *`index`*, a position in the existing list of filters for inserting the new filter can be set. If omitted, *`index`* defaults to the list head (0).

`obj` `object` **`filters clear`**

Removes all filters from *`obj`* and returns the list of removed filters. Clearing is equivalent to passing an empty list for *`filterSpecList`* to `object` **`filter set`**.

`obj` `object` **`filters delete`** **?** `-nocomplain` **?** *`specPattern`*

Removes a single filter from the current list of filters of *`obj`* whose spec matches *`specPattern`*. *`specPattern`* can contain special matching chars (see **`string match`**). `object` **`filters delete`** will throw an error if there is no matching filter, unless `-nocomplain` is set.

`obj` object **`filters get`**

Returns the list of current filter specifications registered for `obj`.

`obj` object **`filters guard`** `methodName` **?** `expr` **?**

If `expr` is specified, registers a guard expression `expr` with a filter `methodName`. This requires that the filter `methodName` has been previously set using object **`filters set`** or added using object **`filters add`**. `expr` must be a valid Tcl expression (see **`expr`**). An empty string for `expr` will clear the currently registered guard expression for filter `methodName`.

If `expr` is omitted, returns the guard expression set on the filter `methodName` defined for `obj`. If none is available, an empty string will be returned.

`obj` object **`filters methods`** **?** `pattern` **?**

If `pattern` is omitted, returns all filter names which are defined by `obj`. By specifying `pattern`, the returned filters can be limited to those whose names match `patterns` (see **`string match`**).

`obj` object **`filters set`** `filterSpecList`

`filterSpecList` takes a list of filter specs, with each spec being itself either a one-element or a two-element list: `methodName` ?-guard `guardExpr`?. `methodName` identifies an existing method of `obj` which becomes registered as a filter. If having three elements, the third element `guardExpr` will be stored as a guard expression of the filter. This guard expression must be a valid Tcl expression (see **`expr`**). `expr` is evaluated when `obj` receives a message to determine whether the filter should intercept the message. Guard expressions allow for realizing context-dependent or conditional filter composition.

Every `methodName` in a `spec` must resolve to an existing method in the scope of the object. To access and to manipulate the list of filters of `obj`, **`cget`** | **`configure`** `-object-filters` can also be used.

forward

`obj` **?** **`public`** **|** **`protected`** **|** **`private`** **?** **`object forward`** `methodName` **?** `-prefix` `prefixName` **? ?** `-frame` object **? ?** `-returns` `valueChecker` **? ?** `-verbose` **? ?** `target` **? ?** `arg` **...?**

Define a forward method for the given object. The definition of a forward method registers a predefined, but changeable list of forwarder arguments under the (forwarder) name `methodName`. Upon calling the forward method, the forwarder arguments are evaluated as a Tcl command call. That is, if present, `target` is interpreted as a Tcl command (e.g., a Tcl **`proc`** or an object) and the remainder of the forwarder arguments `arg` as arguments passed into this command. The actual method arguments to the invocation of the forward method itself are appended to the list of forwarder arguments. If `target` is omitted, the value of `methodName` is implicitly set and used as `target`. This way, when providing a fully-qualified Tcl command name as `methodName` without `target`, the unqualified `methodName` (**`namespace tail`**) is used as the forwarder name; while the fully-qualified one serves as the `target`.

As for a regular **`object method`**, `-returns` allows for setting a value checker on the values returned by the resulting Tcl command call. When passing object to `-frame`, the resulting Tcl command is evaluated in the context of the object receiving the forward method call. This way, variable names used in the resulting execution of a command become resolved as object variables.

The list of forwarder arguments `arg` can contain as its elements a mix of literal values and placeholders. Placeholders are prefixed with a percent symbol (%) and substituted for concrete values upon calling the forward method. These placeholders allow for constructing and for manipulating the arguments to be passed into the resulting command call on the fly:

- `%method` becomes substituted for the name of the forward method, i.e. `methodName`.

- `%self` becomes substituted for the name of the object receiving the call of the forward method.

- `%1` becomes substituted for the first method argument passed to the call of forward method. This requires, in turn, that *at least* one argument is passed along with the method call.

  Alternatively, `%1` accepts an optional argument `defaults`: {`%1` `defaults`}. `defaults` must be a valid Tcl list of two elements. For the first element, `%1` is substituted when there is no first method argument which can be consumed by `%1`. The second element is inserted upon availability of a first method argument with the consumed argument being appended right after the second list element. This placeholder is typically used to define a pair of getter/setter methods.

- {`%@` `index` `value`} becomes substituted for the specified `value` at position `index` in the forwarder-arguments list, with `index` being either a positive integer, a negative integer, or the literal value `end` (such as in Tcl's **`lindex`**). Positive integers specify a list position relative to the list head, negative integers give a position relative to the list tail. Indexes for positioning placeholders in the definition of a forward method are evaluated from left to right and should be used in ascending order.

  Note that `value` can be a literal or any of the placeholders (e.g., `%method`, `%self`). Position prefixes are exempted, they are evaluated as `%` `cmdName`-placeholders in this context.

- {`%argclindex` `list`} becomes substituted for the *n*th element of the provided `list`, with *n* corresponding to the number of method arguments passed to the forward method call.

- `%%` is substituted for a single, literal percent symbol (%).

- `%` `cmdName` is substituted for the value returned from executing the Tcl command `cmdName`. To pass arguments to `cmdName`, the placeholder should be wrapped into a Tcl **`list`**: {`%` `cmdName` ?`arg` ...?}.

  Consider using fully-qualified Tcl command names for `cmdName` to avoid possible name conflicts with the predefined placeholders, e.g., `%self` vs. %**`::nx::self`**.

To disambiguate the names of subcommands or methods, which potentially become called by a forward method, a prefix `prefixName` can be set using `-prefix`. This prefix is prepended automatically to the argument following `target` (i.e., a second argument), if present. If missing, `-prefix` has no effect on the forward method call.

To inspect and to debug the conversions performed by the above placeholders, setting the switch `-verbose` will have the command list to be executed (i.e., after substitution) printed using `::nsf::log` (debugging level: `notice`) upon calling the forward method.

info

`obj` **`info children`** ? `-type` *`className`* ? ? *`pattern`* ?

Retrieves the list of nested (or aggregated) objects of *`obj`*. The resulting list contains the fully qualified names of the nested objects. If `-type` is set, only nested objects which are direct or indirect instances of class *`className`* are returned. Using *`pattern`*, only nested objects whose names match *`pattern`* are returned. The *`pattern`* string can contain special matching characters (see **`string match`**). This method allows for introspecting on **`contains`**.

`obj` **`info class`**

Returns the fully qualified name of the current **`nx::Class`** of *`obj`*. In case of re-classification (see **`configure`**), the returned class will be different from the **`nx::Class`** from which *`obj`* was originally instantiated using **`create`** or **`new`**.

`obj` **`info has`** ? **`mixin`** | **`namespace`** | **`type`** ? ? *`arg`* ...?

`obj` **`info method has mixin`** *`className`*

Verifies whether *`obj`* has a given **`nx::Class`** *`className`* registered as a mixin class (returns: `true`) or not (returns: `false`).

`obj` **`info has namespace`**

Checks whether the object has a companion Tcl namespace (returns: `true`) or not (returns: `false`). The namespace could have been created using, for example, **`object require namespace`**.

`obj` **`info has type`** *`className`*

Tests whether the **`nx::Class`** *`className`* is a type of the object (returns: `true`) or not (returns: `false`). That is, the method checks whether the object is a direct instance of *`className`* or an indirect instance of one of the superclasses of *`className`*.

`obj` **`info lookup`** *`submethod`* ? *`arg`* ...?

A collection of submethods to retrieve structural features (e.g. configuration options, slot objects) and behavioral features (e.g. methods, filters) available for *`obj`* from the perspective of a client to *`obj`*. Features provided by *`obj`* itself and by the classes in its current linearisation list are considered.

`obj` **`info lookup configure parameters`** ? *`namePattern`* ?

Returns all configuration options available for *`obj`* as a list of method-parameter definitions. They can be used, for example, to define a custom method refinement for **`configure`**. The returned configuration options can be limited to those whose names match *`pattern`* (see **`string match`**).

`obj` **`info lookup configure syntax`**

Returns all configuration options available for *`obj`* as a concrete-syntax description to be used in human-understandable messages (e.g. errors or warnings, documentation strings).

obj **info lookup filter** *name*

Returns the method handle for the filter method *name*, if currently registered. If there is no filter *name* registered, an empty string is returned.

obj **info lookup filters** ?`-guards`? ?*namePattern*?

Returns the method handles of all filters which are active on *obj*. By turning on the switch `-guards`, the corresponding guard expressions, if any, are also reported for each filter as a three-element list: *methodHandle* -guard *guardExpr*. The returned filters can be limited to those whose names match *namePattern* (see **string match**).

obj **info lookup method** *name*

Returns the method handle for a method *name* if a so-named method can be invoked on *obj*. If there is no method *name*, an empty string is returned.

obj **info lookup methods** ?*namePattern*?

Returns the names of all methods (including aliases and forwarders) which can be invoked on *obj*. The returned methods can be limited to those whose names match *namePattern* (see **string match**).

obj **info lookup mixins** ?`-guards`? ?*namePattern*?

Returns the object names of all mixin classes which are currently active on *obj*. By turning on the switch `-guards`, the corresponding guard expressions, if any, are also reported as a three-element list for each mixin class: *className* -guard *guardExpr*. The returned mixin classes can be limited to those whose names match *namePattern* (see **string match**).

obj **info lookup slots** ?`-type` *className*? ?`-source` **all | application | system**? ?*namePattern*?

Returns the command names of all slot objects responsible for managing properties, variables, and relations of *obj*. The returned slot objects can be limited according to any or a combination of the following criteria: First, slot objects can be filtered based on their command names matching *namePattern* (see **string match**). Second, `-type` allows one to select slot objects which are instantiated from a subclass *className* of **nx::Slot** (default: **nx::Slot**) . Third, `-source` restricts slot objects returned according to their provenance in either the NX *system* classes or the *application* classes present in the linearisation list of *obj* (default: *all*).

To extract details of each slot object, use the **info** submethods available for each slot object.

obj **info lookup variables**

Returns the command names of all slot objects responsible for managing properties and variables of *obj*, if provided by *obj* or the classes in the linearisation list of *obj*.

This is equivalent to calling: *obj* **info lookup slots** -type ::nx::VariableSlot -source all ?*namePattern*?.

To extract details of each slot object, use the `info` submethods available for each slot object.

`obj` **`info name`**

Returns the unqualified name of an object, i.e., the object name without any namespace qualifiers.

`obj` **`info info`** **?**`-asList`**?**

Returns the available submethods of the **`info`** method ensemble for *`obj`*, either as a pretty-printed string or as a Tcl list (if the switch `-asList` is set) for further processing.

`obj` **`info object filters`** **?**`-guards`**? ?***`pattern`***?**

If *`pattern`* is omitted, returns all filter names which are defined by *`obj`*. By turning on the switch `-guards`, the corresponding guard expressions, if any, are also reported along with each filter as a three-element list: *`filterName`* -guard *`guardExpr`*. By specifying *`pattern`*, the returned filters can be limited to those whose names match *`patterns`* (see **`string match`**).

`obj` **`info object method`** *`option`* *`methodName`*

This introspection submethod provides access to the details of *`methodName`* provided by *`obj`*. Permitted values for *`option`* are:

- `args` returns a list containing the parameter names of *`methodName`*, in order of the method-parameter specification.
- `body` returns the body script of *`methodName`*.
- `definition` returns a canonical command list which allows for (re-)define *`methodName`*.
- `definitionhandle` returns the method handle for a submethod in a method ensemble from the perspective of *`obj`* as method provider. *`methodName`* must contain a complete method path.
- `exists` returns 1 if there is a *`methodName`* provided by *`obj`*, returns 0 otherwise.
- `handle` returns the method handle for *`methodName`*.
- `origin` returns the aliased command if *`methodName`* is an alias method, or an empty string otherwise.
- `parameters` returns the parameter specification of *`methodName`* as a list of parameter names and type specifications.
- `registrationhandle` returns the method handle for a submethod in a method ensemble from the perspective of the method caller. *`methodName`* must contain a complete method path.
- `returns` gives the type specification defined for the return value of *`methodName`*.
- `submethods` returns the names of all submethods of *`methodName`*, if *`methodName`* is a method ensemble. Otherwise, an empty string is returned.

- `syntax` returns the method parameters of `methodName` as a concrete-syntax description to be used in human-understandable messages (e.g., errors or warnings, documentation strings).
- `type` returns whether `methodName` is a *scripted* method, an *alias* method, a *forwarder* method, or a *setter* method.

`obj` **`info object methods`** **?** `-callprotection` `level` **? ?** `-type` `methodType` **? ?** `-path` **? ?** `namePattern` **?**

Returns the names of all methods defined by `obj`. Methods covered include those defined using **`object alias`** and **`object forward`**. The returned methods can be limited to those whose names match `namePattern` (see **`string match`**).

By setting `-callprotection`, only methods of a certain call protection `level` (`public`, `protected`, or `private`) will be returned. Methods of a specific type can be requested using `-type`. The recognized values for `methodType` are:

- `scripted` denotes methods defined using `object` **`method`**;
- `alias` denotes alias methods defined using `object` **`alias`**;
- `forwarder` denotes forwarder methods defined using `object` **`forward`**;
- `setter` denotes methods defined using **`::nsf::setter`**;
- `all` returns methods of any type, without restrictions (also the default value);

`obj` **`info object mixins`** **?** `-guards` **? ?** `pattern` **?**

If `pattern` is omitted, returns the object names of the mixin classes which extend `obj` directly. By turning on the switch `-guards`, the corresponding guard expressions, if any, are also reported along with each mixin as a three-element list: `className` -guard `guardExpr`. The returned mixin classes can be limited to those whose names match `patterns` (see **`string match`**).

`obj` **`info object slots`** **?** `-type` `className` **? ?** `pattern` **?**

If `pattern` is not specified, returns the object names of all slot objects defined by `obj`. The returned slot objects can be limited according to any or a combination of the following criteria: First, slot objects can be filtered based on their command names matching `pattern` (see **`string match`**). Second, `-type` allows one to select slot objects which are instantiated from a subclass `className` of **`nx::Slot`** (default: **`nx::Slot`**).

`obj` **`info object variables`** **?** `pattern` **?**

If `pattern` is omitted, returns the object names of all slot objects provided by `obj` which are responsible for managing properties and variables of `obj`. Otherwise, only slot objects whose names match `pattern` are returned.

This is equivalent to calling: `obj` **`info object slots`** `-type` **`::nx::VariableSlot`** `pattern`.

To extract details of each slot object, use the **`info`** submethods available for each slot object.

`obj` **`info parent`**

Returns the fully qualified name of the parent object of `obj`, if any. If there is no parent object, the name of the Tcl namespace containing `obj` (e.g. "::") will be reported.

`obj` **`info precedence`** **?** `-intrinsic` **? ?** *`pattern`* **?**

Lists the classes from which `obj` inherits structural (e.g. properties) and behavioral features (e.g. methods) and methods, in order of the linearisation scheme in NX. By setting the switch `-intrinsic`, only classes which participate in superclass/ subclass relationships (i.e., intrinsic classes) are returned. If a *`pattern`* is provided only classes whose names match *`pattern`* are returned. The *`pattern`* string can contain special matching characters (see **`string match`**).

`obj` **`info variable`** *`option`* *`handle`*

Retrieves selected details about a variable represented by the given *`handle`*. A *`handle`* can be obtained by querying `obj` using **`info object variables`** and **`info lookup variables`**. Valid values for *`option`* are:

- `name` returns the variable name.
- `parameter` returns a canonical parameter specification eligible to (re-)define the given variable (e.g. using **`object variable`**) in a new context.
- `definition` returns a canonical representation of the definition command used to create the variable in its current configuration.

`obj` **`info vars`** **?** *`pattern`* **?**

Yields a list of Tcl variable names created and defined for the scope of `obj`, i.e., object variables. The list can be limited to object variables whose names match *`pattern`*. The *`pattern`* string can contain special matching characters (see **`string match`**).

method

`obj` **? `public` | `protected` | `private` ? `object method`** *`name`* *`parameters`* **?** `-checkalways` **? ?** `-returns` *`valueChecker`* **?** *`body`*

Defines a scripted method *`methodName`* for the scope of the object. The method becomes part of the object's signature interface. Besides a *`methodName`*, the method definition specifies the method *`parameters`* and a method *`body`*.

`parameters` accepts a Tcl **`list`** containing an arbitrary number of non-positional and positional parameter definitions. Each parameter definition comprises a parameter name, a parameter-specific value checker, and parameter options.

The *`body`* contains the method implementation as a script block. In this body script, the colon-prefix notation is available to denote an object variable and a self call. In addition, the context of the object receiving the method call (i.e., the message) can be accessed (e.g., using **`nx::self`**) and the call stack can be introspected (e.g., using **`nx::current`**).

Optionally, `-returns` allows for setting a value checker on values returned by the method implementation. By setting the switch `-checkalways`, value checking on arguments and return value is guaranteed to be performed, even if value checking is temporarily disabled; see **`nx::configure`**).

A method closely resembles a Tcl `proc`, but it differs in some important aspects: First, a method can define non-positional parameters and value checkers on arguments. Second, the script implementing the method body can contain object-specific notation and commands (see above). Third, method calls *cannot* be intercepted using Tcl `trace`. Note that an existing Tcl `proc` can be registered as an alias method with the object (see `object alias`).

move

obj **move** *newObjectName*

Effectively renames an object. First, the source object *obj* is cloned into a target object *newObjectName* using **copy**. Second, the source object *obj* is destroyed by invoking **destroy**. **move** is also called internally when `rename` is performed for a Tcl command representing an object.

mixins

obj **object mixins** *submethod* ?*arg* ...?

Accesses and modifies the list of mixin classes of *obj* using a specific setter or getter *submethod*:

obj object **mixins add** *spec* ?*index*?

Inserts a single mixin class into the current list of mixin classes of *obj*. Using *index*, a position in the existing list of mixin classes for inserting the new mixin class can be set. If omitted, *index* defaults to the list head (0).

obj object **mixins classes** ?*pattern*?

If *pattern* is omitted, returns the object names of the mixin classes which extend *obj* directly. By specifying *pattern*, the returned mixin classes can be limited to those whose names match *pattern* (see `string match`).

obj object **mixins clear**

Removes all mixin classes from *obj* and returns the list of removed mixin classes. Clearing is equivalent to passing an empty list for *mixinSpecList* to object **mixins set**.

obj object **mixins delete** ?-nocomplain? *specPattern*

Removes a mixin class from a current list of mixin classes of *obj* whose spec matches *specPattern*. *specPattern* can contain special matching chars (see `string match`). object **mixins delete** will throw an error if there is no matching mixin class, unless -nocomplain is set.

obj object **mixins get**

Returns the list of current mixin specifications.

obj object **mixins guard** *className* ?*expr*?

If *expr* is specified, a guard expression *expr* is registered with the mixin class *className*. This requires that the corresponding mixin class *className* has been previously set using object **mixins set** or added using object **mixins add**. *expr* must be a valid Tcl expression (see

`expr`). An empty string for *`expr`* will clear the currently registered guard expression for the mixin class *`className`*.

If *`expr`* is not specified, returns the active guard expression. If none is available, an empty string will be returned.

`obj` `object` **`mixins set`** *`mixinSpecList`*

`mixinSpecList` represents a list of mixin class specs, with each spec being itself either a one-element or a three-element list: *`className`* ?-guard *`guardExpr`*?. If having one element, the element will be considered the *`className`* of the mixin class. If having three elements, the third element *`guardExpr`* will be stored as a guard expression of the mixin class. This guard expression will be evaluated using **`expr`** when *`obj`* receives a message to determine if the mixin is to be considered during method dispatch or not. Guard expressions allow for realizing context-dependent or conditional mixin composition.

At the time of setting the mixin relation, that is, calling `object` **`mixins`**, every *`className`* as part of a spec must be an existing instance of **`nx::Class`**. To access and to manipulate the list of mixin classes of *`obj`*, **`cget`** | **`configure`** `-object-mixins` can also be used.

__object_configureparameter

`obj` **`__object_configureparameter`**

Computes and returns the configuration options available for *`obj`*, to be consumed as method-parameter specification by **`configure`**.

property

`obj` **`object property`** **?** `-accessor` `public` **|** `protected` **|** `private` **? ?** `-configurable` *`trueFalse`* **? ?** `-incremental` **? ?** `-class` *`className`* **? ?** `-nocomplain` **?** *`spec`* **?** *`initBlock`* **?**

Defines a property for the scope of the object. The *`spec`* provides the property specification as a **`list`** holding at least one element or, maximum, two elements: *`propertyName`* ? **:** *`typeSpec`* ? ? *`defaultValue`* ?. The *`propertyName`* is also used as to form the names of the getter/setter methods, if requested (see `-accessor`). It is, optionally, equipped with a *`typeSpec`* following a colon delimiter which specifies a value checker for the values which become assigned to the property. The second, optional element sets a *`defaultValue`* for this property.

If `-accessor` is set, a property will provide for a pair of getter and setter methods:

`obj` *`propertyName`* **`set`** *`value`*

Sets the property *`propertyName`* to *`value`*.

`obj` *`propertyName`* **`get`**

Returns the current value of property *`propertyName`*.

`obj` *`propertyName`* **`unset`**

Removes the value store of *`propertyName`* (e.g., an object variable), if existing.

The option value passed along `-accessor` sets the level of call protection for the generated getter and setter methods: `public`, `protected`, or `private`. By default, no getter and setter methods are created.

Turning on the switch `-incremental` provides a refined setter interface to the value managed by the property. First, setting `-incremental` implies requesting `-accessor` (set to `public` by default, if not specified explicitly). Second, the managed value will be considered a valid Tcl list. A multiplicity of `1..*` is set by default, if not specified explicitly as part of *spec*. Third, to manage this list value element-wise (*incrementally*), two additional setter methods become available:

obj *propertyName* **add** *element* **?** *index* **?**

Adding *element* to the managed list value, at the list position given by *index* (by default: 0).

obj *propertyName* **delete** *elementPattern*

Removing one or multiple elements from the managed list value which match *elementPattern*. *elementPattern* can contain matching characters (see **string match**).

By setting `-configurable` to `true` (the default), the property can be accessed and modified through **cget** and **configure**, respectively. If `false`, no configuration option will become available via **cget** and **configure**.

If neither `-accessor` nor `-configurable` are requested, the value managed by the property will have to be accessed and modified directly. If the property manages an object variable, its value will be readable and writable using **set** and **eval**.

A property becomes implemented by a slot object under any of the following conditions:

- `-configurable` equals `true` (by default).
- `-accessor` is one of `public`, `protected`, or `private`.
- `-incremental` is turned on.
- *initBlock* is a non-empty string.

Assuming default settings, every property is realized by a slot object.

Provided a slot object managing the property is to be created, a custom class *className* from which this slot object is to be instantiated can be set using `-class`. The default value is **::nx::VariableSlot**.

The last argument *initBlock* accepts an optional Tcl script which is passed into the initialization procedure (see **configure**) of the property's slot object. See also *initBlock* for **create** and **new**.

By default, the property will ascertain that no (potentially) pre-existing and equally named object variable will be overwritten when defining the property. In case of a conflict, an error exception is thrown:

```
% Object create obj { set :x 1 }
::obj
% ::obj object property {x 2}
object ::obj has already an instance variable named 'x'
```

If the switch `-nocomplain` is on, this check is omitted (continuing the above example):

```
% ::obj object property -nocomplain {x 2}
% ::obj eval {set :x}
2
```

require

`obj` **`require namespace`**

Create a Tcl namespace named after the object *`obj`*. All object variables become available as namespace variables.

`obj` **`require`** **?** **`public`** **|** **`protected`** **|** **`private`** **?** **`object method`** *`methodName`*

Attempts to register a method definition made available using **`::nsf::method::provide`** under the name *`methodName`* with *`obj`* . The registered method is subjected to default call protection (`protected`), if not set explicitly.

unknown

`obj` **`unknown`** *`unknownMethodName`* **?** *`arg`* **...?**

This method is called implicitly whenever an unknown method is invoked. *`unknownMethodName`* indicates the unresolvable method name, followed by the remainder of the original argument vector as a number of *`arg`* of the indirected method invocation.

variable

`obj` **`object variable`** **?** `-accessor` `public` **|** `protected` **|** `private` **? ?** `-incremental` **? ?** `-class` *`className`* **? ?** `-configurable` *`trueFalse`* **? ?** `-initblock` *`script`* **? ?** `-nocomplain` **?** *`spec`* **?** *`defaultValue`* **?**

Defines a variable for the scope of the object. The *`spec`* provides the variable specification: *`variableName`* ? **:** *`typeSpec`* ?. The *`variableName`* will be used to name the underlying Tcl variable and the getter/setter methods, if requested (see `-accessor`). *`spec`* is optionally equipped with a *`typeSpec`* following a colon delimiter which specifies a value checker for the values managed by the variable. Optionally, a *defaultValue* can be defined.

If `-accessor` is set explicitly, a variable will provide for a pair of getter and setter methods:

`obj` *`variableName`* **`set`** *`varValue`*

Sets *`variableName`* to *`varValue`* .

`obj` *`variableName`* **`get`**

Returns the current value of *`variableName`* .

`obj` *`variableName`* **`unset`**

Removes *`variableName`* , if existing, underlying the property.

The option value passed along `-accessor` sets the level of call protection for the getter and setter methods: `public`, `protected`, or `private`. By default, no getter and setter methods are created.

Turning on the switch `-incremental` provides a refined setter interface to the value managed by the variable. First, setting `-incremental` implies requesting `-accessor` (`public` by default, if not specified explicitly). Second, the managed value will be considered a valid Tcl list. A multiplicity of `1..*` is set by default, if not specified explicitly as part of *`spec`* (see above). Third, to manage this list value element-wise (*incrementally*), two additional setter operations become available:

`obj` *`variableName`* **`add`** *`element`* **?** *`index`* **?**

Adding *`element`* to the managed list value, at the list position given by *`index`* (by default: 0).

`obj` *`variableName`* **`delete`** *`elementPattern`*

Removing one or multiple elements from the managed list value which match *`elementPattern`*. *`elementPattern`* can contain matching characters (see **`string match`**).

By setting `-configurable` to `true`, the variable can be accessed and modified via **`cget`** and **`configure`**, respectively. If `false` (the default), the interface based on **`cget`** and **`configure`** will not become available. In this case, and provided that `-accessor` is set, the variable can be accessed and modified via the getter/setter methods. Alternatively, the underlying Tcl variable, which is represented by the variable, can always be accessed and modified directly, e.g., using **`eval`**. By default, `-configurable` is `false`.

A variable becomes implemented by a slot object under any of the following conditions:

- `-configurable` equals `true`.
- `-accessor` is one of `public`, `protected`, or `private`.
- `-incremental` is turned on.
- `-initblock` is a non-empty string.

Provided a slot object managing the variable is to be created, a custom class *`className`* from which this slot object is to be instantiated can be set using `-class`. The default value is **`::nx::VariableSlot`**.

Using `-initblock`, an optional Tcl *`script`* can be defined which becomes passed into the initialization procedure (see **`configure`**) of the variable's slot object. See also *`initBlock`* for **`create`** and **`new`**.

By default, the variable will ascertain that a pre-existing and equally named object variable will not be overwritten when defining the variable. In case of a conflict, an error exception is thrown:

```
% Object create obj { set :x 1 }
::obj
% ::obj object variable x 2
object ::obj has already an instance variable named 'x'
```

If the switch `-nocomplain` is on, this check is omitted (continuing the above example):

```
% ::obj object variable -nocomplain x 2
% ::obj eval {set :x}
2
```

OBJECT SELF-REFERENCE

Objects are naturally recursive, with methods of an object `::obj` frequently invoking other methods in the same object `::obj` and accessing `::obj`'s object variables. To represent these self-references effectively in method bodies, and dependening on the usage scenario, NX offers two alternative notations for self-references: one based on a special-purpose syntax token ("colon prefix"), the other based on the command `nx::current`.

Both, the colon-prefix notation and `nx::current`, may be used only in method bodies and scripts passed to **eval**. If they appear anywhere else, an error will be reported. There are three main use cases for self-references:

1. As a *placeholder* for the currently active object, `nx::current` can be used to retrieve the object name.

2. Reading and writing *object variables* directly (i.e. without getter/setter methods in place) require the use of variable names carrying the prefix `:` ("colon-prefix notation"). Internally, colon-prefixed variable names are processed using Tcl's variable resolvers. Alternatively, one can provide for getter/setter methods for object variables (see **property** and **variable**).

3. *Self-referential method calls* can be defined via prefixing (`:`) the method names or, alternatively, via `nx::current`. Internally, colon-prefixed method names are processed using Tcl's command resolvers. The colon-prefix notation is recommended, also because it has a (slight) performance advantage over `nx::current` which requires two rather than one command evaluation per method call.

See the following listing for some examples corresponding to use cases 1--3:

```
Object create ::obj {
  puts [current];                         # 1) print name of currently active object ('::obj
  set :x 1; :object variable y 2;        # 2) object variables
  :public object method print {} {
    set z 3;                                 # 2.a) method-local variable
    puts ${:x}-${:y}-$z;                 # 2.b) variable substitution using '$' and ':'
    puts [set :x]-[set :y]-[set z];         # 2.c) reading variables using 'set'
    set :x 1; incr :y;                         # 2.d) writing variables using 'set', 'incr'
  }
  :public object method show {} {
   :print;                                # 3.a) self-referential method call using ':'
   [current] print;                        # 3.b) self-referential method call using 'nx::cu
   [current object] print;              # 3.c) self-referential method call using 'nx::curre
  }
  :show
}
```

COPYRIGHT