nx::Class(3) 2.0 Class ""

NAME

nx::Class - API reference of the base-metaclass of the NX objectsystem

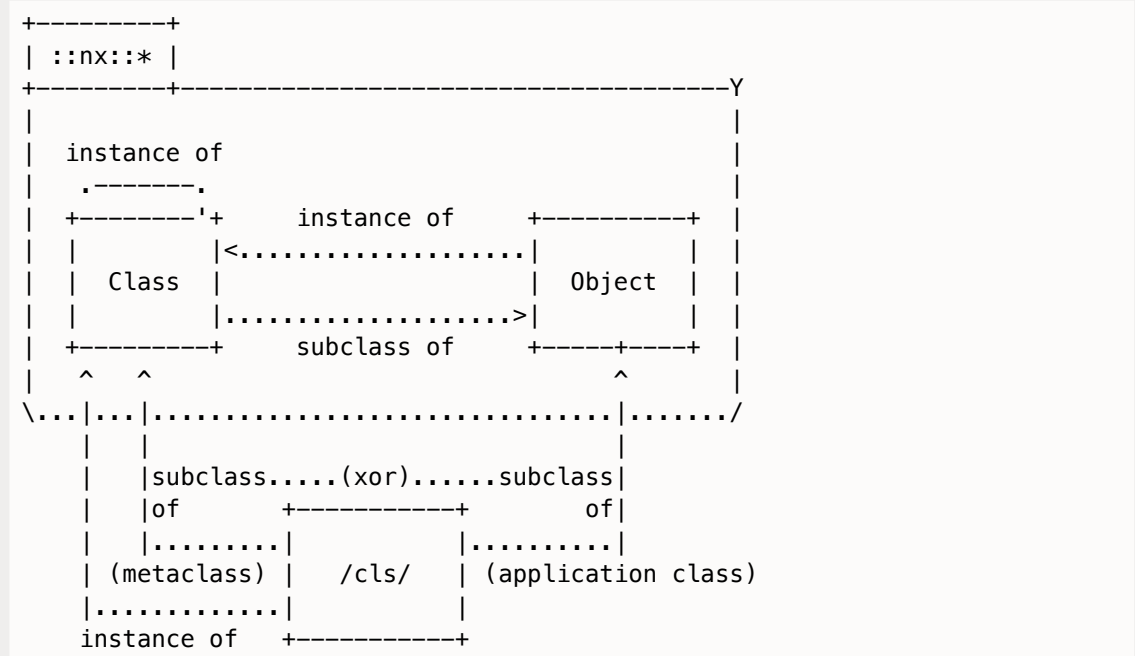
TABLE OF CONTENTS

SYNOPSIS

nx::Class create *cls* ?**-superclasses** *superClassNames*? ?**-mixins** *mixinSpec*? ?**-filters** *filterSpec*? ?*option value ...*? ?*initBlock*?
nx::Class new ?**-superclasses** *superClassNames*? ?**-mixins** *mixinSpec*? ?**-filters** *filterSpec*? ?*initBlock*?
cls ?**public** | **private** | **protected**? **alias** *methodName* ?**-returns** *valueChecker*? ?**-frame object** | **method**? *cmdName*
cls **create** *instanceName* ?*option value option value ...*?
cls **delete** *feature arg*
cls **filters** *submethod* ?*arg ...*?
cls ?**public** | **protected** | **private**? **forward** *methodName* ?**-prefix** *prefixName*? ?**-frame object**? ?**-returns** *valueChecker*? ?**-verbose**? ?*target*? ?*arg ...*?
cls **info heritage** ?*pattern*?
cls **info instances** ?**-closure**? ?*pattern*?
cls **info mixinof** ?**-closure**? ?**-scope** *option*? ?*pattern*?
cls **info subclasses** ?**-closure**? ?**-dependent**? ?*pattern*?
cls **info superclasses** ?**-closure**? ?*pattern*?
cls **info info** ?**-asList**?
cls **info filters** ?**-guards**? ?*pattern*?
cls **info method** *option methodName*
cls **info methods** ?**-callprotection** *level*? ?**-type** *methodType*? ?**-path**? ?*namePattern*?
cls **info mixins** ?**-guards**? ?*pattern*?
cls **info slots** ?**-type** *className*? ?*pattern*?
cls **info variables** ?*pattern*?
cls ?**public** | **protected** | **private**? **method** *name parameters* ?**-checkalways**? ?**-returns** *valueChecker*? *body*
cls **mixins** *submethod* ?*arg ...*?
cls **new** ?**-childof** *parentName*? ?*option value option value ...*?
cls **property** ?**-accessor public** | **protected** | **private**? ?**-configurable** *trueFalse*? ?**-incremental**? ?**-class** *className*? *spec* ?*initBlock*?
cls **require** ?**public** | **protected** | **private**? **method** *methodName*
cls **variable** ?**-accessor public** | **protected** | **private**? ?**-incremental**? ?**-class** *className*? ?**-configurable** *trueFalse*? ?**-initblock** *script*? *spec* ?*defaultValue*?

DESCRIPTION

nx::Class is the base metaclass of the NX object system. All classes (e.g. *cls*) are (direct or indirect) instances of **nx::Class**. Therefore, the methods provided by **nx::Class** are available to all classes. A class *cls* which does not have **nx::Class** as its direct or indirect superclass is referred to as an *application class*. By default, when instantiating a new class from **nx::Class**, it becomes an application class with **nx::Object** being set as its superclass. A class *cls* which is explicitly declared as a (direct or indirect) subclass of **nx::Class** is referred to as a *metaclass*, that is, its instances will become classes as well. In other words, a metaclass instantiates and subclasses **nx::Class** at the same time.



Classes can be created in the following ways:

nx::Class **create** *cls* **?** -superclasses *superClassNames* **? ?** -mixins *mixinSpec* **? ?** -filters *filterSpec* **? ?** *option* *value* **...? ?** *initBlock* **?**

To create a class having the explicit name *cls*, use **create**.

nx::Class **new** **?** -superclasses *superClassNames* **? ?** -mixins *mixinSpec* **? ?** -filters *filterSpec* **? ?** *initBlock* **?**

To create a class having an automatically assigned, implicit name, use **new**.

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

CONFIGURATION OPTIONS FOR INSTANCES OF NX::CLASS

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**.

`-superclasses` **?** *`superClassNames`* **?**

If *`superClassNames`* is not specified, returns the superclasses of the class. If provided, the class becomes the subclass of *`superClassNames`*.

`-filters` **?** *`filterSpecs`* **?**

Retrieves the list of filter methods currently active on instances of the class, if *`filterSpecs`* is not set. Otherwise, activates a list of filter methods for the instances of the class. Filters are returned or set in terms of a list of filter specifications.

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

Returns the list of mixin classes currently active on instances of the class, if *`mixinSpecs`* is not specified. Otherwise, the class is extended by the list of mixin classes provided by *`mixinSpecs`*. mixin classes are returned or set in terms of a list of mixin specifications.

The configuration options provided by **`nx::Object`** are equally available because an application class *cls* is an indirect instance of **`nx::Object`**.

METHODS FOR INSTANCES OF NX::CLASS

alias

`cls` **?** **`public`** | **`private`** | **`protected`** **?** **`alias`** *`methodName`* **?** `-returns` *`valueChecker`* **?** **?** `-frame` `object` | `method` **?** *`cmdName`*

Define an alias method for the given class. The resulting method registers a pre-existing Tcl command *`cmdName`* under the (alias) name *`methodName`* with the class. 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 **`class 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.

__class_configureparameter

`cls` **`__class_configureparameter`**

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

create

cls **create** *instanceName* **?***option value option value* **...?**

This factory method creates an instance *instanceName* of *cls* and returns *instanceName*.

```
% nx::Class create AClass {
    :method init args {
      next
    }; # initialization method for instances of 'AClass'
  }; # defines a class 'AClass' being an instance of 'nx::Class'
::AClass
% ::AClass create anInstance; # defines an object 'anInstance' being an instance of
::anInstance
% ::anInstance info class
::AClass
% ::AClass info class
::nx::Class
```

create accepts the configuration options *option* available for this instance, such as those defined by properties of *cls* (see **property**).

Note that **create** is called internally when defining an instance of *cls* using **new**.

By calling **create** on **nx::Class** itself, the created instance will become a new application class *instanceName* on which **create** can also be applied (i.e., it can be instantiated). If the so-created class has **::nx::Class** its direct or indirect superclass, *instanceName* is referred to as a metaclass; that is, a class whose instances are again classes.

delete

cls **delete** *feature arg*

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

cls **delete property** *propertyName*

cls **delete variable** *variableName*

cls **delete method** *methodName*

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

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

filters

cls **filters** *submethod* **?***arg* **...?**

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

cls **filters add** *spec* **?***index***?**

Inserts a single filter into the current list of filters of *cls*. 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).

cls **filters clear**

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

cls **filters delete ?**`-nocomplain`**?** *specPattern*

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

cls **filters get**

Returns the list of current filter specifications registered for *cls*.

cls **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 **filters set** or added using **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 *cls*. If none is available, an empty string will be returned.

cls **filters methods ?***pattern***?**

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

cls **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 *cls* 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 *cls* 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 class. To access and to manipulate the list of filters of *cls*, **cget** | **configure** `-filters` can also be used.

forward

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

Define a forward method for the given class. 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 **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

A collection of introspection submethods on the structural features (e.g. configuration options, superclasses) and the behavioral features (e.g. methods, filters) provided by *cls* to its instances.

`cls` **`info heritage`** **?** *`pattern`* **?**

If *`pattern`* is omitted, returns the list of object names of all the direct and indirect superclasses and *per-class* mixin classes of *`cls`*, in their order of precedence, which are active for instances of *`cls`*. If *`pattern`* is specified, only superclasses and mixin classes whose names match *`pattern`* will be listed (see **`string match`**).

`cls` **`info instances`** **?** `-closure` **? ?** *`pattern`* **?**

If *`pattern`* is not specified, returns a list of the object names of all the direct instances of *`cls`*. If the switch `-closure` is set, indirect instances are also returned. A direct instance is created by using **`create`** or **`new`** on *`cls`*, an indirect instance was created from a direct or indirect subclass of *`cls`*. If *`pattern`* is specified, only instances whose names match *`pattern`* will be listed (see **`string match`**).

`cls` **`info mixinof`** **?** `-closure` **? ?** `-scope` *`option`* **? ?** *`pattern`* **?**

If *`pattern`* is not specified, returns a list of the object names of all the objects for which *`cls`* is active as a direct mixin class. If the switch `-closure` is set, objects which have *`cls`* as an indirect mixin class are also returned. If *`pattern`* is specified, only objects whose names match *`pattern`* will be listed (see **`string match`**). Valid values of *`option`* are `all`, `object`, and `class`. Passing `object` will have only objects returned which have *`cls`* as *per-object* mixin class. Passing `class` will have only classes returned which have *`cls`* as *per-class* mixin class. `all` (the default) will have contained both in the returned list.

`cls` **`info subclasses`** **?** `-closure` **?** **?** `-dependent` **?** **?** *`pattern`* **?**

If *`pattern`* is not specified, returns a list of the object names of the direct subclasses of *`cls`*. If the switch `-closure` is set, indirect subclasses are also returned. If the switch `-dependent` is on, indirect subclasses introduced by mixin class relations of subclasses of *`cls`* are also reported. `-closure` and `-dependent` are mutually exclusive. If *`pattern`* is specified, only subclasses whose names match *`pattern`* will be listed (see **`string match`**).

`cls` **`info superclasses`** **?** `-closure` **?** **?** *`pattern`* **?**

If *`pattern`* is not specified, returns a list of the object names of all direct superclasses of *`cls`*. If the switch `-closure` is set, indirect superclasses will also be returned. If *`pattern`* is specified, only superclasses whose names match *`pattern`* will be listed (see **`string match`**).

`cls` **`info info`** **?** `-asList` **?**

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

`cls` **`info filters`** **?** `-guards` **?** **?** *`pattern`* **?**

If *`pattern`* is omitted, returns all filter names which are defined by *`cls`*. 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`**).

`cls` **`info method`** *`option`* *`methodName`*

This introspection submethod provides access to the details of *`methodName`* provided by *`cls`*. 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 *`cls`* as method provider. *`methodName`* must contain a complete method path.
- `exists` returns 1 if there is a *`methodName`* provided by *`cls`*, 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.

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

Returns the names of all methods defined by *`cls`*. Methods covered include those defined using **`alias`** and **`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 `class` **`method`**;
- `alias` denotes alias methods defined using `class` **`alias`**;
- `forwarder` denotes forwarder methods defined using `class` **`forward`**;
- `setter` denotes methods defined using **`::nsf::setter`**;
- `all` returns methods of any type, without restrictions (also the default value);

`cls` **`info mixins`** **`?`** `-guards` **`?`** **`?`** *`pattern`* **`?`**

If *`pattern`* is omitted, returns the object names of the mixin classes which extend *`cls`* 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`**).

`cls` **`info slots`** **`?`** `-type` *`className`* **`?`** **`?`** *`pattern`* **`?`**

If *`pattern`* is not specified, returns the object names of all slot objects defined by *`cls`*. 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`**).

`cls` **`info variables`** **?**`pattern`**?**

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

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

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

method

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

Defines a scripted method `methodName` for the scope of the class. The method becomes part of the class'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 class (see **`alias`**).

mixins

`cls` **`mixins`** `submethod` **?**`arg` **...?**

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

`cls` **`mixins add`** `spec` **?**`index`**?**

Inserts a single mixin class into the current list of mixin classes of `cls`. 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).

cls **mixins classes** ?*pattern*?

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

cls **mixins clear**

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

cls **mixins delete** ?-nocomplain? *specPattern*

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

cls **mixins get**

Returns the list of current mixin specifications.

cls **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 `class` **mixins set** or added using **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.

cls **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 *cls* 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 **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 *cls*, **cget** | **configure** `-mixins` can also be used.

new

cls **new** ?`-childof` *parentName*? ?*option* *value* *option* *value* ...?

A factory method to create autonamed instances of *cls*. It returns the name of the newly created instance. For example:

```
% nx::Class create AClass; # defines a class 'AClass' being an instance of 'nx::Cla
::AClass
% set inst [::AClass new]; # defines an autonamed object being an instance of 'ACla
::nsf::__#0
% $inst info class
::AClass
```

The factory method will provide computed object names of the form, e.g. `::nsf::__#0`. The uniqueness of generated object names is guaranteed for the scope of the current Tcl interpreter only.

It is a frontend to **`create`** which will be called by **`new`** once the name of the instance has been computed, passing along the arguments *`option`* to **`new`** as the configuration options (see **`create`**).

If `-childof` is provided, the new object will be created as a nested object of *`parentName`*. *`parentName`* can be the name of either an existing NX object or an existing Tcl namespace. If non-existing, a Tcl namespace *`parentName`* will be created on the fly.

property

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

Defines a property for the scope of the class. 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**.

require

cls **require** **?** **public** **|** **protected** **|** **private** **?** **method** *methodName*

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

variable

cls **variable** **?** -accessor public **|** protected **|** private **?** **?** -incremental **?** **?** -class *className* **?** **?** -configurable *trueFalse* **?** **?** -initblock *script* **?** *spec* **?** *defaultValue* **?**

Defines a variable for the scope of the class. 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:

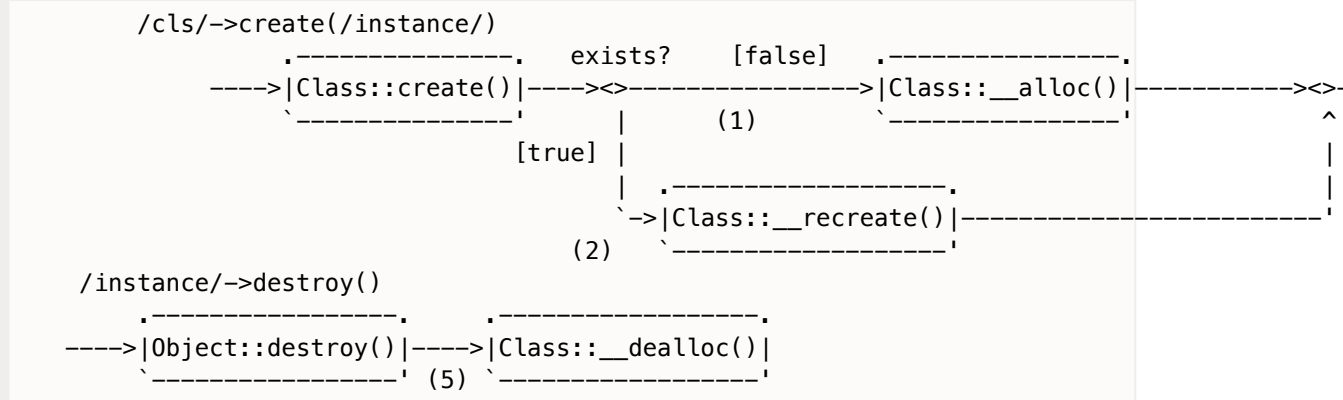
- `-configurable` equals `true`.
- `-accessor` is one of `public`, `protected`, or `private`.
- `-incremental` is turned on.
- `-initblock` is a non-empty string.

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**.

OBJECT LIFE CYCLE

nx::Class provides means to control important stages through which an NX object passes between and including its creation and its destruction: allocation, recreation, deallocation.



Object creation is controlled by the factory method **create**, provided by **nx::Class** to its instance *cls*. **create** produces a new object *instance* as an instance of *cls* in a number of steps.

1. If *instance* does not represent an existing object, an internal call to **__alloc**, provided by **nx::Class**, runs the *allocation* procedure for a fresh *instance* of *cls*.

2. If *instance* corresponds to an existing object, the *recreation* procedure is triggered by calling **__recreate** defined by **nx::Class**.

3. The newly allocated or recreated object *instance* is then configured by dispatching **configure**, provided by **nx::Object**, which consumes the configuration options passed into **create**. This will establish the instance's initial state, e.g. by setting object variables and object relations according to the configuration options and corresponding default values.

4. Finally, the initialization method **init** is dispatched, if available for *instance*. **init** can be defined by *cls* on behalf of its instance *instance*, e.g. to lay out a class-specific initialisation behaviour.

   ```
   % nx::Class create Foo {:property x}
   % Foo method init {} {set :y [expr {${:x} + 1}]}
   % Foo public method bar {} {return ${:y}}
   % Foo create f1 -x 101
   % f1 cget -x
   101
   % f1 bar
   102
   ```

   Alternatively, the object *instance* may define an per-object **init** on its own. A per-object **init** can be chained to a class-level **init** using **nx::next**, just like a regular method.

   Note that the definition of an **init** method must contain an empty parameter specification, since **init** is always called with an empty argument list.

Object destruction, such as triggered by an application-level **destroy** call (5), is finalized by **__dealloc** offerd by **nx::Class**.

In the following, the three built-in procedures --- allocation, recreation, and deallocation --- are explained:

- *Allocation*: **__alloc** creates a blank object *instance* as an instance of *cls* and returns the fully-qualified *instance*. **__alloc** is primarily used internally by **create** to allocate a Tcl memory storage for *instance* and to register *instance* with the Tcl interpreter as a new command.

- *Recreation*: Recreation is the NX scheme for resolving naming conflicts between objects: An object is requested to be created using **create** or **new** while an object of an identical object name, e.g. *instance*, already exists:

  ```
  % Object create Bar
  ::Bar
  % Object create Bar; # calls Object->__recreate(::Bar, ...)
  ::Bar
  ```

  In such a situation, the built-in **__recreate** first unsets the object state (i.e., Tcl variables held by the object) and removes relations of the object under recreation with other objects. Then, second, standard object initialization is performed by calling **configure** and **init**, if any.

  Alternatively, recreation will be performed as a sequence of **destroy** and **create** calls in the following recreation scenarios:

  - An existing class is requested to be recreated as an object.

  - An existing object is requested to be recreated as a class.

    ```
    % Object create Bar
    ::Bar
    % Class create Bar; # calls Bar->destroy() & Class::create(::Bar, ...)
    ```

  - An object of an object system other than NX (e.g. XOTcl2) is asked to be recreated.

- *Deallocation*: **__dealloc** marks an instance *instance* of *cls* for deletion by returning its Tcl memory representation to the Tcl memory pool and by unregistering the corresponding Tcl command with the Tcl interpreter.

  Beware that **__dealloc** does not necessarily cause the object to be deleted immediately. Depending on the lifecycle of the object's environment (e.g. the Tcl interp interpreter, the containing namespace) and on call references down the callstack, the actual memory freeing/returning operation may occur at a later point.

The three methods **__alloc**, **__recreate**, and **__dealloc** are internally provided and internally called. By default, they are not part of the method interface of *cls* and cannot be called directly by clients of *cls*. In addition, **__alloc**, **__recreate**, and **__dealloc** are protected from redefinition by a script.

To extend or to replace the built-in allocation, recreation, and deallocation procedure, the methods **__alloc**, **__recreate**, and **__dealloc** can be refined by providing a custom method implementation:

- as a per-object method of *cls*;

- as a method of a per-object mixin class extending *cls*;

- as a method of a per-class mixin class extending `nx::Class`;

- as a method of a subclass specializing **nx::Class**, from which *cls* is to be instantiated.

This custom implementation can redirect to the built-in **__alloc**, **__recreate**, and **__dealloc**, respectively, by using **nx::next**. By providing such a custom implementation, **__alloc**, **__recreate**, and **__dealloc**, respectively, become available as callable methods of *cls*:

cls **__alloc** *instance*

cls **__recreate** *instance* **?** *arg* **...?**

cls **__dealloc** *instance*

COPYRIGHT