Migration Guide for the Next Scripting Language

Gustaf Neumann
<neumann@wu-wien.ac.at>
version 2.3.0, May 2019

Table of Contents

JavaScript must be enabled in your browser to display the table of contents.

Abstract

This document describes the differences between the Next Scripting Language Framework and XOTcl 1. In particular, it presents a migration guide from XOTcl 1 to NX, and presents potential incompatibilities between XOTcl 1 and XOTcl 2.

The Next Scripting Language (NX) is a successor of XOTcl 1 and is based on 10 years of experience with XOTcl in projects containing several hundert thousand lines of code. While XOTcl was the first language designed to provide language support for design patterns, the focus of the Next Scripting Framework and NX are on combining this with Language Oriented Programming. In many respects, NX was designed to ease the learning of the language by novices (by using a more mainstream terminology, higher orthogonality of the methods, less predefined methods), to improve maintainability (remove sources of common errors) and to encourage developer to write better structured programs (to provide interfaces) especially for large projects, where many developers are involved.

The Next Scripting Language is based on the Next Scripting Framework which was developed based on the notion of language oriented programming. The Next Scripting Frameworks provides C-level support for defining and hosting multiple object systems in a single Tcl interpreter. The whole definition of NX is fully scripted (e.g. defined in `nx.tcl`). The Next Scripting Framework is shipped with three language definitions, containing NX and XOTcl 2. Most of the existing XOTcl 1 programs can be used without modification in the Next Scripting Framework by using XOTcl 2. The Next Scripting Framework requires Tcl 8.5 or newer.

Although NX is fully scripted (as well as XOTcl 2), our benchmarks show that scripts based on NX are often 2 or 4 times faster than the counterparts in XOTcl 1. But speed was not the primary focus on the Next Scripting Environment: The goal was primarily to find ways to repackage the power of XOTcl in an easy to learn environment, highly orthogonal environment, which is better suited for large projects, trying to reduce maintenance costs.

We expect that many user will find it attractive to upgrade from XOTcl 1 to XOTcl 2, and some other users will upgrade to NX. This document focuses mainly on the differences between XOTcl 1 and NX, but addresses as well potential incompatibilities between XOTcl 1 and XOTcl 2. For an introduction to NX, please consult the NX tutorial.

1. Differences Between XOTcl and NX

The Next Scripting Framework supports *Language Oriented Programming* by providing means to define potentially multiple object systems with different naming and functionality in a single interpreter. This makes the Next Scripting Framework a powerful instrument for defining multiple languages such as e.g. domain specific languages. This focus differs from XOTcl 1.

Technically, the language framework approach means that the languages implemented by the Next

Scripting Framework (most prominently XOTcl 2 and NX) are typically fully scripted and can be loaded via the usual Tcl `package require` mechanism.

Some of the new features below are provided by the Next Scripting Framework, some are implemented via the script files for XOTcl 2 and NX.

1.1. Features of NX

In general, the Next Scripting Language (NX) differs from XOTcl in the following respects:

1. **Stronger Encapsulation:** The Next Scripting Language favors a *stronger form of encapsulation* than XOTcl. Calling the own methods or accessing the own instance variables is typographically easier and computationally faster than these operations on other objects. This behavior is achieved via *resolvers*, which make some methods necessary in XOTcl 1 obsolete in NX (especially for importing instance variables). The encapsulation of NX is stronger than in XOTcl but still weak compared to languages like C++; a developer can still access other objects' variables via some idioms, but NX *makes accesses to other objects' variables explicit*. The requiredness to make these accesses explicit should encourage developer to implement well defined interfaces to provide access to instance variables.
2. **Additional Forms of Method Definition and Reuse:** The Next Scripting Language provides much more orthogonal means to *define, reuse and introspect* scripted and C-implemented methods.
    a. It is possible to use NX `alias` to register methods under arbitrary names for arbitrary objects or classes.
    b. NX provides means for *method protection* (method modifiers `public`, `protected`, and `private`). Therefore developers have to define explicitly public interfaces in order to use methods from other objects.
    c. One can invoke in NX fully qualified methods to invoke methods outside the precedence path.
    d. One can define in NX *hierarchical method names* (similar to commands and subcommands, called method ensembles) in a convenient way to provide extensible, hierarchical naming of methods.
    e. One can use in NX the same interface to query (introspect) C-implemented and scripted methods/commands.
3. **Orthogonal Parameterization:** The Next Scripting Language provides an *orthogonal framework for parametrization* of methods and objects.
    a. In NX, the same argument parser is used for
        - Scripted Methods
        - C-implemented methods and Tcl commands
        - Object Parametrization
    b. While XOTcl 1 provided only value-checkers for non-positional arguments for methods, the Next Scripting Framework provides the same value checkers for positional and non-positional arguments of methods, as well as for positional and non-positional configure parameters (`-parameter` in XOTcl 1).
    c. While XOTcl 1 supported only non-positional arguments at the begin of the argument list, these can be used now at arbitrary positions.
4. **Value Checking:**
    a. The Next Scripting Language supports checking of the *input parameters* and the *return values* of scripted and C-implemented methods and commands.

   b. NX provides a set of predefined checkers (like e.g. `integer`, `boolean`, `object`, ...) which can be extended by the applications.
   c. Value Checking can be used for *single* and *multi-valued* parameters. One can e.g. define a list of integers with at least one entry by the parameter specification `integer,1..n`.
   d. Value Checking can be turned on/off globally or on the method/command level.
5. **Scripted Init Blocks:** The Next Scripting Language provides *scripted init blocks* for objects and classes (replacement for the dangerous dash "-" mechanism in XOTcl that allows one to set variables and invoke methods upon object creation).
6. **More Conventional Naming for Predefined Methods:** The naming of the methods in the Next Scripting Language is much more in line with the mainstream naming conventions in OO languages. While for example XOTcl uses `proc` and `instproc` for object specific and inheritable methods, NX uses simply `method`.
7. **Profiling Support:** The Next Scripting Language provides now two forms of profiling
   - Profiling via a DTrace provider (examples are e.g. in the dtrace subdirectory of the source tree)
   - Significantly improved built-in profiling (results can be processed in Tcl).
8. **Significantly Improved Test Suite:** The regression test suite of Next Scripting Scripting framework contain now more than 5.000 tests, and order of magnitude more than in XOTcl 1.6
9. **Much Smaller Interface:** The Next Scripting Language has a much *smaller interface* (i.e. provides less predefined methods) than XOTcl (see Table 1), although the expressiveness was increased in NX.

Table 1. Comparison of the Number of Predefined Methods in NX and XOTcl

	NX	XOTcl
Methods for Objects	14	51
Methods for Classes	9	24
Info-methods for Objects	11	25
Info-methods for Classes	11	24
Total	**45**	**124**

This comparison list compares mostly XOTcl 1 with NX, some features are also available in XOTcl 2 (2a, 2c 2d, 3, 4).

1.2. NX and XOTcl Scripts

Below is a small, introductory example showing an implementation of a class `Stack` in NX and XOTcl. The purpose of this first example is just a quick overview. We will go into much more detailed comparison in the next sections.

NX supports a block syntax, where the methods are defined during the creation of the class. The XOTcl syntax is slightly more redundant, since every definition of a method is a single toplevel command starting with the class name (also NX supports the style used in XOTcl). In NX, all methods are per

default protected (XOTcl does not support protection). In NX methods are defined in the definition of the class via `:method` or `:public method`. In XOTcl methods are defined via the `instproc` method.

Another difference is the notation to refer to instance variables. In NX, instance variable are named with a single colon in the front. In XOTcl, instance variables are imported using `instvar`.

Stack example in NX

```
Class create Stack {

   #
   # Stack of Things
   #

   :variable things ""

   :public method push {thing} {
      set :things [linsert ${:things} 0 $thing]
      return $thing
   }

   :public method pop {} {
      set top [lindex ${:things} 0]
      set :things [lrange ${:things} 1 end]
      return $top
   }
}
```

Stack example in XOTcl

```
#
# Stack of Things
#

Class Stack

Stack instproc init {} {
   my instvar things
   set things ""
}

Stack instproc push {thing} {
   my instvar things
   set things [linsert $things 0 $thing]
   return $thing
}

Stack instproc pop {} {
   my instvar things
   set top [lindex $things 0]
   set things [lrange $things 1 end]
}
```

1.3. Using XOTcl 2.0 and the Next Scripting Language in a Single Interpreter

In general, the Next Scripting Framework supports multiple object systems concurrently. Effectively, every object system has different base classes for creating objects and classes. Therefore, these object systems can have different interfaces and names of built-in methods. Currently, the Next Scripting Framework is packaged with three object systems:

- NX
- XOTcl 2.0
- TclCool

XOTcl 2 is highly compatible with XOTcl 1, the language NX is described below in more details, the language TclCool was introduced in Tip#279 and serves primarily an example of a small OO language.

A single Tcl interpreter can host multiple Next Scripting Object Systems at the same time. This fact makes migration from XOTcl to NX easier. The following example script shows to use XOTcl and NX in a single script:

Using Multiple Object Systems in a single Script

```
namespace eval mypackage {

  package require XOTcl 2.0

  # Define a class with a public method foo using XOTcl
  xotcl::Class C1
  C1 instproc foo {} {puts "hello world"}

  package require nx

  # Define a class with a public method foo using NX
  nx::Class create C2 {
    :public method foo {} {puts "hello world"}
  }
}
```

One could certainly create object or classes from the different object systems via fully qualified names (e.g. using e.g. `::xotcl::Class` or `::nx::Class`), but for migration for systems without explicit namespaces switching between the object systems eases migration. "Switching" between XOTcl and NX effectively means the load some packages (if needed) and to import either the base classes (Object and Class) of XOTcl or NX into the current namespace.

2. XOTcl Idioms in the Next Scripting Language

The following sections are intended for reader familiar with XOTcl and show, how certain language Idioms of XOTcl can be expressed in NX. In some cases, multiple possible realizations are listed

2.1. Defining Objects and Classes

When creating objects or classes, one should use the method `create` explicitly. In XOTcl, a default `unknown` method handler was provided for classes, which create for every unknown method invocation an object/class with the name of the invoked method. This technique was convenient, but as well dangerous, since typos in method names lead easily to unexpected behavior. This default unknown method handler is not provided in NX (but can certainly be provided as a one-liner in NX by the application).

XOTcl	Next Scripting Language
`Class ClassName`	`Class create ClassName`
`Object ObjectName`	`Object create ObjectName`

2.2. Defining Methods

In general, both XOTcl and NX support methods on the object level (per-object methods, i.e. methods only applicable to a single object) and on the class level (methods inherited to instances of the classes). While the naming in XOTcl tried to follow closely the Tcl tradition (using the term `proc` for functions/methods), NX uses the term `method` for defining scripted methods.

XOTcl uses the prefix `inst` to denote that methods are provided for instances, calling therefore scripted methods for instances `instproc`. This is certainly an unusual term. The approach with the name prefix has the disadvantage, that for every different kind of method, two names have to be provided (e.g. `proc` and `instproc`, `forward` and `instforward`).

NX on the contrary uses the same term for defining instance method or object-specific methods. When the term (e.g. `method`) is used on a class, the method will be an instance method (i.e. applicable to the instances of the class). When the term is used on an object with the modifier `object`, an object-specific method is defined. This way one can define the same way object specific methods on an object as well as on a class.

Furthermore, both XOTcl and NX distinguish between scripted methods (section 3.2.1) and C-defined methods (section 3.2.2). Section 3.2.3 introduces method protection, which is only supported by NX.

2.2.1. Scripted Methods Defined in the Init-block of a Class/Object or with

Separate Calls

The following examples show the definition of a class and its methods in the init-block of a class (NX only), and the definition of methods via separate top level calls (XOTcl and NX).

XOTcl

```
# Define instance method 'foo' and object
# method 'bar' for a Class 'C' with separate
# toplevel commands

Class C
C instproc foo args {...}
C proc bar args {...}
```

Next Scripting Language

```
# Define instance method and object method
# in the init-block of a class

Class create C {
  :method foo args {...}
  :object method bar args {...}
}
```

```
# Define instance method and object method
# with separate commands

Class create C
C method foo args {...}
C object method bar args {...}
```

```
# Define object-specific method foo
# for an object 'o' with separate commands

Object o
o set x 1
o proc foo args {...}
```

```
# Define object method and set
# instance variable in the init-block of
# an object

Object create o {
  set :x 1
  :object method foo args {...}
}
```

```
# Define object method and set
# instance variable with separate
# commands

Object create o
o eval {set :x 1}
o object method foo args {...}
```

2.2.2. Different Kinds of Methods

This section describes various kinds of methods. The different kinds of methods are defined via different method-defining methods, which are summarized in the following table for XOTcl and NX.

XOTcl

```
# Methods for defining methods:
#
#     proc
#     instproc
#     forward
#     instforward
#     parametercmd
#     instparametercmd
#
# All these methods return empty.
```

Next Scripting Language

```
# Methods for defining methods:
#
#     alias
#     forward
#     method
#
# All these methods return method-handles.
```

In addition to scripted methods (previous section) XOTcl supports forwarder (called `forward` and `instforward`) and accessor functions to variables (called `parametercmd` and `instparametercmd`). The accessor functions are used normally internally when object-specific parameters are defined (see Section 3.4).

In NX forwarders are called `forward`. NX does not provide a public available method to define variable accessors like `parametercmd` in XOTcl, but use internally the Next Scripting Framework primitive

`nsf::method::setter` when appropriate.

XOTcl

```
Class C
C instforward f1 ...
C forward f2 ...

Object o
o forward f3 ...
```

Next Scripting Language

```
# Define forwarder

Class create C {
  :forward f1 ...
  :object forward f2 ...
}

Object create o {
  :object forward f3 ...
}
```

XOTcl

```
# Define setter and getter methods in XOTcl.
#
# XOTcl provides methods for these.

Class C
C instparametercmd p1
C parametercmd p2

Object o
o parametercmd p3
```

Next Scripting Language

```
# Define setter and getter methods in NX.
#
# NX does not provide own methods, but uses
# the low-level framework commands, since
# application developer will only
# need it in rare cases.

Class create C
::nsf::method::setter C p1
::nsf::method::setter C -per-object p2

Object create o
::nsf::method::setter o p3
```

NX supports in contrary to XOTcl the method `alias` which can be used to register arbitrary Tcl commands or methods for an object or class under a provided method name. Aliases can be used to reuse a certain implementation in e.g. different object systems under potentially different names. In some respects aliases are similar to forwarders, but they do not involve forwarding overhead.

XOTcl

```
# Method "alias" not available
```

Next Scripting Language

```
# Define method aliases
# (to scripted or non-scripted methods)

Class create C {
  :alias a1 ...
  :object alias a2 ...
}

Object create o {
  :object alias a3 ...
}
```

2.2.3. Method Modifiers and Method Protection

NX supports four method modifiers `object`, `public`, `protected` and `private`. All method modifiers can be written in front of every method defining command. The method modifier `object` is used to denote object-specific methods (see above). The concept of method protection is new in NX.

XOTcl

```
# Method modifiers
#
#   "object",
#   "public",
#   "protected", and
#   "private"
#
# are not available
```

Next Scripting Language

```
# Method modifiers
#
#   "object",
#   "public",
#   "protected"
#
# are applicable for all kinds of
# method defining methods:
```

XOTcl

Next Scripting Language

```
#
#    method, forward, alias
#
# The modifier "private" is available for
#
#    method, forward, alias
#
Class create C {
  :/method-definition-method/ ...
  :public /method-definition-method/ ...
  :protected /method-definition-method/ ...
  :private /method-definition-method/ ...
  :object /method-definition-method/ ...
  :public object /method-definition-method/ ...
  :protected object /method-definition-method/ ...
  :private object /method-definition-method/ ...
}
```

XOTcl does not provide method protection. In NX, all methods are defined per default as protected. This default can be changed by the application developer in various ways. The command `::nx::configure defaultMethodCallProtection true|false` can be used to set the default call protection for scripted methods, forwarder and aliases. The defaults can be overwritten also on a class level.

NX provides means for method hiding via the method modifier `private`. Hidden methods can be invoked only via the `-local` flag, which means: "call the specified method defined in the same class/object as the currently executing method".

XOTcl

```
# XOTcl provides no means for
# method hiding
```

Next Scripting Language

```
# Hiding of methods via "private"
#
nx::Class create Base {
  :private method baz {a b} {expr {$a + $b}}
  :public method foo {a b} {: -local baz $a $b}
}

nx::Class create Sub -superclass Base {
  :public method bar {a b} {: -local baz $a $b}
  :private method baz {a b} {expr {$a * $b}}

  :create s1
}

s1 foo 3 4  ;# returns 7
s1 bar 3 4  ;# returns 12
s1 baz 3 4  ;# unable to dispatch method 'baz'
```

2.2.4. Method Deletion

NX provides an explicit `delete` method for the deletion of methods.

XOTcl

```
# XOTcl provides only method deletion with
# the equivalent of Tcl's "proc foo {} {}"
/cls/ instproc foo {} {}
/obj/ proc foo {} {}
```

Next Scripting Language

```
# Deletion of Methods
#
/cls/ delete method /name/
/obj/ delete object method /name/
```

2.3. Resolvers

The Next Scripting Framework defines Tcl resolvers for method and variable names to implement object specific behavior. Within the bodies of scripted methods these resolver treat variable and function names starting with a colon `:` specially. In short, a colon-prefixed variable name refers to an instance variable, and a colon-prefixed function name refers to a method. The sub-sections below provide detailed examples.

Note that the resolvers of the Next Scripting Framework can be used in the XOTcl 2.* environment as well.

2.3.1. Invoking Methods

In XOTcl, a method of the same object can be invoked via `my`, or in general via using the name of the object in front of the method name.

In NX, the own methods are called via the method name prefixed with a single colon. The invocation of the methods of other objects is the same in NX and XOTcl.

XOTcl

```
Class C
C instproc foo args {...}
C instproc bar args {
  my foo 1 2 3 ;# invoke own method
  o baz        ;# invoke other object's method
}
Object o
o proc baz {} {...}
```

Next Scripting Language

```
Class create C {
  :method foo args {...}
  :method bar args {
     :foo 1 2 3 ;# invoke own method
     o baz      ;# invoke other object's method
  }
}
Object create o {
  :public object method baz {} {...}
}
```

2.3.2. Accessing Own Instance Variables from Method Bodies

In general, the Next Scripting Language favors the access to an objects's own instance variables over variable accesses of other objects. This means that in NX it is syntactically easier to access the own instance variables. On the contrary, in XOTcl, the variable access to own and other variables are fully symmetric.

In XOTcl, the following approaches are used to access instance variables:

- Import instance variables via `instvar` and access variables via `$varName`
- Set or get instance variables via `my set varName ?value?` or other variable accessing methods registered on `xotcl::Object` such as `append`, `lappend`, `incr`, etc.
- Register same-named accessor functions and set/get values of instance variables via `my varName ?value?`

In NX, the favored approach to access instance variables is to use the name resolvers, although it is as well possible to import variables via `nx::var import` or to check for the existence of instance variables via `nx::var exists`.

The following examples summary the use cases for accessing the own and other instance variables.

XOTcl

```
Class C
```

Next Scripting Language

```
Class create C {
```

XOTcl

```
C instproc foo args {
  # Method scoped variable a
  set a 1
  # Instance variable b
  my instvar b
  set b 2
  # Global variable/namespaced variable c
  set ::c 3
}
```

```
... instproc ... {
   my set /varName/ ?value?
}
```

```
... instproc ... {
   my instvar /varName/
   set /varName/ ?value?
}
```

```
... instproc ... {
   set /varName/ [my set /otherVar/]
}
```

```
... instproc ... {
   my exists /varName/
}
```

Next Scripting Language

```
  :method foo args {...}
    # Method scoped variable a
    set a 1
    # Instance variable b
    set :b 2
    # Global variable/namespaced variable c
    set ::c 3
  }
}
```

```
# Set own instance variable to a value via
# resolver (preferred and fastest way)

... method ... {
   set :/newVar/ ?value?
}
```

```
# Set own instance variable via
# variable import

... method ... {
   ::nx::var import [self] /varName/
   set /varName/ ?value?
}
```

```
# Read own instance variable

... method ... {
   set /varName/ [set :/otherVar/]
}
```

```
... method ... {
   set /newVar/ ${:/otherVar/}
}
```

```
# Test existence of own instance variable

... method ... {
   info :/varName/
}
```

```
 ... method ... {
   ::nx::var exists [self] /varName/
}
```

2.3.3. Accessing Instance Variables of other Objects

XOTcl

```
/obj/ set /varName/ ?value?
```

```
set /varName/ [/obj/ set /otherVar/]
```

Next Scripting Language

```
# Set instance variable of object obj to a
# value via resolver
# (preferred way: define property on obj)

/obj/ eval [list set :/varName/ ?value?]
```

```
# Read instance variable of object obj
# via resolver
```

XOTcl

```
... instproc ... {
   /obj/ instvar /varName/
   set /varName/ ?value?
}
```

Next Scripting Language

```
set /varName/ [/obj/ eval {set :/otherVar/}]
```

```
# Read instance variable of object /obj/
# via import

... method ... {
   ::nx::var import /obj/ /varName/
   set /varName/ ?value?
}
```

```
/obj/ exists varName
```

```
# Test existence of instance variable of
# object obj

/obj/ eval {info exists :/varName/}
```

```
::nx::var exists /obj/ /varName/
```

2.4. Parameters

While XOTcl 1 had very limited forms of parameters, XOTcl 2 and NX provide a generalized and highly orthogonal parameter machinery handling various kinds of value constraints (also called value checkers). Parameters are used to specify,

- how objects and classes are initialized (we call these parameter types *Configure Parameters*), and
- what values can be passed to methods (we call these *Method Parameters*).

Furthermore, parameters might be positional or non-positional, they might be optional or required, they might have a defined multiplicity, and value-types, they might be introspected, etc. The Next Scripting Framework provide a unified, C-implemented infrastructure to handle both, object and method parameters in the same way with a high degree of orthogonality.

Configuration parameters were specified in XOTcl 1 primarily via the method `parameter` in a rather limited way, XOTcl 1 only supported non-positional parameters in front of positional ones, supported no value constraints for positional parameters, provided no distinction between optional and required, and did not support multiplicity.

Furthermore, the Next Scripting Framework provides optionally *Return Value Checking* based on the same mechanism to check whether some methods return always the values as specified.

2.4.1. Parameters for Configuring Objects: Variables and Properties

Configure parameters are used for specifying values for configuring objects when they are created (i.e. how instance variables are initialized, what parameters can be passed in for initialization, what default values are used, etc.). Such configuration parameters are supported in XOTcl primarily via the method `parameter`, which is used in XOTcl to define multiple parameters via a list of parameter specifications.

Since the term "parameter" is underspecified, NX uses a more differentiated terminology. NX distinguishes between configurable instance variables (also called *properties*) and non configurable instance variables (called *variables*), which might have as well e.g. default values. The values of configurable properties can be queried at runtime via `cget`, and their values can be altered via `configure`. When the value of a configure parameter is provided or changed, the value checkers from the variable definition are used to ensure, the value is permissible (i.e. it is for example an integer value).

The sum of all configurable object parameters are called *configure parameters*. To define a define a configurable variable, NX uses the method `property`, for non-configurable variables, the method `variable` is used.

Optionally, one can define in NX, that a `property` or a `variable` should have a public, protected or private accessor. Such an accessor is a method with the same name as the variable. In XOTcl, every `parameter` defined as well automatically a same-named accessor method, leading to potential name conflicts with other method names.

In the examples below we show the definition of configurable a non-configurable instance variables using `variable` and `property` respectively.

XOTcl

```
# Define class "Foo" with instance
# variables "x" and "y" initialized
# on instance creation. The initialization
# has to be performed in the constructor.

Class Foo
Foo instproc init args {
   instvar x y
   set x 1
   set y 2
}

# Create instance of the class Foo
Foo f1

# Object f1 has instance variables
# x == 1 and y == 2
```

Next Scripting Language

```
# Define class "Foo" with instance variables
# "x" and "y" initialized on instance creation.
# The method "variable" is similar in syntax
# to Tcl's "variable" command. During
# instance creation, the variable
# definitions are used for the
# initialization of the variables of the object.

Class create Foo {
  :variable x 1
  :variable y 2
}

# Create instance of the class Foo
Foo create f1

# Object f1 has instance variables
# x == 1 and y == 2
```

While XOTcl follows a procedural way to initialize variables via the constructor `init`, NX follows a more declarative approach. Often, classes have superclasses, which often want to provide their own instance variables and default values. The declarative approach from NX solves this via inheritance, while a procedural approach via assign statements in the constructor requires explicit constructor calls, which are often error-prone. Certainly, when a user prefers to assign initial values to instance variables via explicit assign operations in constructors, this is as well possible in NX.

NX uses the same mechanism to define class variables or object variables.

XOTcl

```
# No syntactic support for creating
# class variables
```

Next Scripting Language

```
# Define a object variable "V" with value 100 and
# an instance variable "x". "V" is defined for the
# class object Foo, "x" is defined in the
# instances of the class. "object variable" works
# similar to "object method".

Class create Foo {
  :object variable V 100
  :variable x 1
}
```

In the next step, we define configurable instance variables which we call *properties* in NX.

XOTcl uses the method `parameter` is a shortcut for creating multiple configurable variables with automatically created accessors (methods for reading and writing of the variables). In NX, the preferred way to create configurable variables is to use the method `property`. The method `property` in NX is similar to `variable`, but makes the variables configurable, which means that

1. one can specify the property as a non-positional parameter upon creation of the object,
2. one can query the value via the method `cget`, and

3. one can modify the value of the underlying variable via the method `configure`.

XOTcl

```
# Parameters specified as a list
# (short form); parameter
# "a" has no default, "b" has default "1"

Class Foo -parameter {a {b 1} {c "[info
tclversion]"}}

# Create instance of the class Foo
Foo f1 -a 0

# Object f1 has instance variables
# a == 0 and b == 1

# XOTcl registers automatically accessors
# for the parameters. Use the accessor
# "b" to output the value of variable "b"
puts [f1 b]

# Use the setter to alter value of
# instance variable "b"
f1 b 100

# Return the substituted value of
# parameter "c", something like 8.7.
# XOTcl substitutes always when it sees
# square brackets or dollar signs.
f1 c
```

Next Scripting Language

```
# Define property "a" and "b". The
# property "a" has no default, "b" has
# default value "1"

Class create Foo {
  :property a
  :property {b 1}
  :property {c "[info tclversion]"}
  :property {d:substdefault "[info tclversion]"}
}

# Create instance of the class Foo
Foo create f1 -a 0

# Object f1 has instance variables
# a == 0 and b == 1

# Use the method "cget" to query the value
# of a configuration parameter
puts [f1 cget -b]

# Use the method "configure" to alter the
# value of instance variable "b"
f1 configure -b 100

# Return the (non substituted) value of
# parameter "c", and the substituted value
# of parameter "d"
f1 cget -c
f1 cget -d
```

In general, NX allows one to create variables and properties with and without accessor methods. The created accessor methods might be `public`, `protected` or `public`. When the value `none` is provided to `-accessor`, no accessor will be created. This is actually the default in NX. In order to change the default behavior in NX, one can use `::nx::configure defaultAccessor none|public|protected|private`.

XOTcl

```
# "parameter" creates always accessor
# methods, accessor methods are
# always public, no "cget" is available.

Class create Foo -parameter {a {b 1}}

# Use the accessor method to query
# the value of a configuration parameter
puts [f1 b]

# Use the accessor method to set the
# value of instance variable "a"
f1 a 100

# Use the accessor method to unset the
# value of instance variable "a" n.a. via
# accessor
```

Next Scripting Language

```
# Define property "a" and "b". The
# property "a" has no default, "b" has
# default value "1"

Class create Foo {
  :variable -accessor public a
  :property -accessor public {b 1}
}

# Use the accessor method to query
# the value of a configuration parameter
puts [f1 b get]

# Use the accessor method to set the
# value of instance variable "a"
f1 a set 100

# Use the accessor method to unset the
# value of instance variable "a"
f1 a unset
```

Similar to `variable`, properties can be defined in NX on the class and on the object level.

XOTcl

```
# XOTcl provides no means to define
# configurable variables at the object
# level
```

Next Scripting Language

```
# Define class with a property for the class object
# named "cp". This is similar to "static variables"
# in some other object-oriented programming
# languages.

Class create Foo {
  ...
  :object property cp 101
}

# Define object property "op"

Object create o {
  :object property op 102
}
```

NX supports *value constraints* (value-checkers) for object and method parameters in an orthogonal manner. NX provides a predefined set of value checkers, which can be extended by the application developer. In NX, the *value checking is optional*. This means that it is possible to develop e.g. which a large amount of value-checking and deploy the script with value checking turned off, if the script is highly performance sensitive.

XOTcl

```
# No value constraints for
# parameter available
```

Next Scripting Language

```
# Predefined value constraints:
#    object, class, alnum, alpha, ascii, boolean,
#    control, digit, double, false, graph, integer,
#    lower, parameter, print, punct, space, true,
#    upper, wordchar, xdigit
#
# User defined value constraints are possible.
# All parameter value checkers can be turned on
# and off at runtime.
#
# Define a required boolean property "a"
# and an integer property "b" with a default.
# The first definition uses "properties",
# the second definition uses multiple
# "property" statements.

Class create Foo -properties {
   a:boolean
   {b:integer 1}
}
```

```
Class create Foo {
   :property a:boolean
   :property {b:integer 1}
}
```

In XOTcl all configure parameters were *optional*. Required parameters have to be passed to the constructor of the object.

NX allows one to define *optional* and *required* configure parameters (as well as method parameters). Therefore, configure parameters can be used as the single mechanism to parametrize objects. It is in NX not necessary (and per default not possible) to pass arguments to the constructor.

XOTcl

```
# Required parameter not available
```

Next Scripting Language

```
# Required parameter:
# Define a required property "a" and a
# required boolean property "b"

Class create Foo -properties {
```

XOTcl

Next Scripting Language

```
   a:required
   b:boolean,required
}
```

```
Class create Foo {
   :property a:required
   :property b:boolean,required
}
```

NX supports in contrary to XOTcl to define the *multiplicity* of values per parameter. In NX, one can specify that a parameter can accept the value "" (empty) in addition to e.g. an integer, or one can specify that the value is an empty or non-empty list of values via the multiplicity. For every specified value, the value checkers are applied.

XOTcl

```
# Multiplicity for parameter
# not available
```

Next Scripting Language

```
# Parameter with multiplicity
#   ints is a list of integers, with default
#   objs is a non-empty list of objects
#   obj is a single object, maybe empty

Class create Foo -properties {
  {ints:integer,0..n ""}
   objs:object,1..n
   obj:object,0..1
}
```

```
Class create Foo {
  :property {ints:integer,0..n ""}
  :property objs:object,1..n
  :property obj:object,0..1
}
```

For the implementation of variables and properties, NX uses slot objects, which are an extension to the `-slots` already available in XOTcl. While very for every `property` in NX, a slot object is created, for performance reasons, not every `variable` has a slot associated.

When a property is created, NX does actually three things:

1. Create a slot object, which can be specified in more detail using the init-block of the slot object
2. Create a parameter definition for the initialization of the object (usable via a non-positional parameter during object creation), and
3. register optionally an accessor function (setter), for which the usual protection levels (`public`, `protected` or `private`) can be used.

XOTcl

```
# Define parameters via slots

Class Foo -slots {
   Attribute a
   Attribute b -default 1
}

# Create instance of the class Foo
# and provide a value for instance
# variable "a"
Foo f1 -a 0

# Object f1 has a == 0 and b == 1
```

Next Scripting Language

```
# Configurable parameters specified via the
# method "property" (supports method
# modifiers and scripted configuration;
# see below)

Class create Foo {
   :property a
   :property {b 1}
}

# Create instance of the class Foo and
# provide a value for instance variable "a"
Foo create f1 -a 0
```

XOTcl

Next Scripting Language

```
# Object f1 has a == 0 and b == 1
```

Since the slots are objects, the slot objects can be configured and parametrized like every other object in NX. Slot objects can be provided with a scripted initialization as well. We show first the definition of properties similar to the functionality provided as well by XOTcl and show afterwards how to use value constraints, optional parameters, etc. in NX.

XOTcl

```
# Define parameter with an
# attribute-specific type checker

Class Person -slots {
  Attribute create sex -type "sex" {
    my proc type=sex {name value} {
      switch -glob $value {
        m* {return m}
        f* {return f}
        default {
          error "expected sex but got $value"
        }
      }
    }
  }
}
```

Next Scripting Language

```
# Configure parameter with scripted
# definition (init-block), defining a
# property specific type checker

Class create Person {
    :property -accessor public sex:sex,convert {

      # define a converter to standardize
representation
      :object method type=sex {name value} {
        switch -glob $value {
          m* {return m}
          f* {return f}
          default {error "expected sex but got
$value"}
        }
      }

    }
}
```

The parameters provided by a class for the initialization of instances can be introspected via querying the parameters of the method create: `/cls/ info lookup parameters create` (see [info_configure_parameter]).

2.4.2. Delete Variable Handlers

XOTcl

Next Scripting Language

```
# No syntactic support for deleting
# variable handlers
```

```
# Like deletion of Methods:
# Delete on the object, where the
# variable handler is defined.

/cls/ delete property /name/
/obj/ delete object property /name/

/cls/ delete variable /name/
/obj/ delete object variable /name/
```

2.4.3. Method Parameters

Method parameters are used to specify the interface of a single method (what kind of values may be passed to a method, what default values are provided etc.). The method parameters specifications in XOTcl 1 were limited and allowed only value constraints for non positional arguments.

NX and XOTcl 2 provide value constraints for all kind of method parameters. While XOTcl 1 required non-positional arguments to be listed in front of positional arguments, this limitation is lifted in XOTcl 2.

XOTcl

```
# Define method foo with non-positional
# parameters (x, y and y) and positional
# parameter (a and b)

Class C
C instproc foo {
   -x:integer
   -y:required
   -z
   a
   b
} {
   # ...
}
C create c1

# invoke method foo
c1 foo -x 1 -y a 2 3
```

Next Scripting Language

```
# Define method foo with
# non-positional parameters
# (x, y and y) and positional
# parameter (a and b)

Class create C {
   :public method foo {
      -x:integer
      -y:required
      -z
      a
      b
   } {
      # ...
   }
   :create c1
}
# invoke method foo
c1 foo -x 1 -y a 2 3
```

```
# Only leading non-positional
# parameters are available; no
# optional positional parameters,
# no value constraints on
# positional parameters,
# no multiplicity, ...
```

```
# Define various forms of parameters
# not available in XOTcl 1

Class create C {
  # trailing (or interleaved) non-positional
  # parameters
  :public method m1 {a b -x:integer -y} {
    # ...
  }

  # positional parameters with value constraints
  :public method m2 {a:integer b:boolean} {
    #...
  }

  # optional positional parameter (trailing)
  :public method set {varName value:optional} {
    # ....
  }

  # parameter with multiplicity
  :public method m3 {-objs:object,1..n
c:class,0..1} {
    # ...
  }

  # In general, the same list of value
  # constraints as for configure parameter is
  # available (see above).
  #
  # User defined value constraints are
  # possible. All parameter value checkers
  # can be turned on and off.
}
```

2.4.4. Return Value Checking

Return value checking is a functionality available in the Next Scripting Framework, that was not yet available in XOTcl 1. A return value checker assures that a method returns always a value satisfying some value constraints. Return value checkers can be defined on all forms of methods (scripted or C-implemented). Like for other value checkers, return value checkers can be turned on and off.

XOTcl

```
# No return value checking
# available
```

Next Scripting Language

```
# Define method foo with non-positional
# parameters (x, y and y) and positional
# parameter (a and b)

Class create C {
```

XOTcl

Next Scripting Language

```
  # Define method foo which returns an
  # integer value
  :method foo -returns integer {-x:integer} {
    # ...
   }

  # Define an alias for the Tcl command ::incr
  # and assure, it always returns an integer
  # value
  :alias incr -returns integer ::incr

  # Define a forwarder that has to return an
  # integer value
  :forward ++ -returns integer ::expr 1 +

 # Define a method that has to return a
 # non-empty list of objects
 :public object method instances {} \
    -returns object,1..n {
   return [:info instances]
  }
}
```

2.5. Interceptors

XOTcl and NX allow the definition of the same set of interceptors, namely class- and object-level mixins and class- and object-level filters. The primary difference in NX is the naming, since NX abandons the prefix "inst" from the names of instance specific method, but uses the modifier `object`" for object specific methods.

Therefore, in NX, if a `mixin` is registered on a class-level, it is applicable for the instances (a per-class mixin), and if and `object mixin` is registered, it is a per-object mixin. In both cases, the term `mixin` is used, in the second case with the modifier `object`. As in all other cases, one can register the same way a per-object mixin on a plain object or on a class object.

2.5.1. Register Mixin Classes and Mixin Guards

XOTcl

```
/cls/ instmixin ...
/cls/ instmixinguard /mixin/ ?condition?

# Query per-class mixin
/cls/ instmixin
```

```
/obj/ mixin ...
/obj/ mixinguard /mixin/ ?condition?

# Query per-object mixins
/obj/ mixin
```

Next Scripting Language

```
# Register/clear per-class mixin and guard for
# a class

/cls/ mixins add|set|clear ...
/cls/ mixins guard /mixin/ ?condition?
/cls/ configure -mixin ...

# Query per-class mixins
/cls/ mixins get
/cls/ cget -mixins

# Query per-class mixins (without guards)
/cls/ mixins classes
```

```
# Register/clear per-object mixin and guard for
# an object

/obj/ object mixins add|set|clear ...
/obj/ object mixins guard /mixin/ ?condition?
/obj/ configure -object-mixins ...

# Query per-object mixin
/obj/ object mixins get
```

XOTcl

Next Scripting Language

```
/obj/ cget -object-mixin

# Query per-object mixins (without guards)
/cls/ mixins classes
```

2.5.2. Register Filters and Filter Guards

XOTcl

```
# Register per-class filter and guard for
# a class
/cls/ instfilter ...
/cls/ instfilterguard /filter/ ?condition?

# Query per-class filter
/cls/ instfilter
```

Next Scripting Language

```
# Register/clear per-class filter and guard for
# a class

/cls/ filters add|set|clear ...
/cls/ filters guard /filter/ ?condition?
/cls/ configure -filters ...

# Query per-class filters
/cls/ filters get
/cls/ cget -filters

# Query per-class filters (without guards)
/cls/ filters methods
```

```
/obj/ filter ...
/obj/ filterguard /filter/ ?condition?
```

```
# Register(clear per-object filter and guard for
# an object

/obj/ object filters add|set|clear ...
/obj/ object filters guard /filter/ ?condition?
/obj/ configure -object-filters ...

# Query per-object filters
/cls/ object filters get
/obj/ cget -object-filters

# Query per-object filters (without guards)
/cls/ object filters methods
```

2.6. Introspection

In general, introspection in NX became more orthogonal and less dependent on the type of the method. In XOTcl it was e.g. necessary that a developer had to know, whether a method is e.g. scripted or not and has to use accordingly different sub-methods of `info`.

In NX, one can use e.g. always `info method` with a subcommand and the framework tries to hide the differences as far as possible. So, one can for example obtain with `info method parameter` the parameters of scripted and C-implemented methods the same way, one can get the definition of all methods via `info method definition` and one can get an manual-like interface description via `info method syntax`. In addition, NX provides means to query the type of a method, and NX allows one to filter by the type of the method.

2.6.1. List sub- and superclass relations

While XOTcl used singular words for introspecting sub- and superclass relations, NX uses plural word to indicate that potentially a list of values is returned.

XOTcl	Next Scripting Language
`/cls/ info superclass ?pattern?`	`/cls/ info superclasses ?pattern?`
`/cls/ info subclass ?pattern?`	`/cls/ info subclasses -type setter ?pattern?`

2.6.2. List methods defined by classes

While XOTcl uses different names for obtaining different kinds of methods defined by a class, NX uses `info methods` in an orthogonal manner. NX allows as well to use the call protection to filter the returned methods.

XOTcl	Next Scripting Language
`/cls/ info instcommands ?pattern?`	`/cls/ info methods ?pattern?`
`/cls/ info instparametercmd ?pattern?`	`/cls/ info methods -type setter ?pattern?`
`/cls/ info instprocs ?pattern?`	`/cls/ info methods -type scripted ?pattern?`

`# n.a.`	`/cls/ info methods -type alias ?pattern?`
	`/cls/ info methods -type forwarder ?pattern?`
	`/cls/ info methods -type object ?pattern?`
	`/cls/ info methods -callprotection public|protected ...`

2.6.3. List methods defined by objects

While XOTcl uses different names for obtaining different kinds of methods defined by an object, NX uses `info methods` in an orthogonal manner. NX allows as well to use the call protection to filter the returned methods.

XOTcl	Next Scripting Language
`/obj/ info commands ?pattern?`	`/obj/ info object methods ?pattern?`
`/obj/ info parametercmd ?pattern?`	`/obj/ info object methods -type setter ?pattern?`
`/obj/ info procs ?pattern?`	`/obj/ info object methods -type scripted ?pattern?`

`# n.a.`	`/obj/ info object methods -type alias ?pattern?`
	`/obj/ info object methods -type forwarder ?pattern?`
	`/obj/ info object methods -type object ?pattern?`
	`/obj/ info object methods -callprotection public|protected ...`

2.6.4. Check existence of a method

NX provides multiple ways of checking, whether a method exists; one can use `info method exists` to check, if a given method exists (return boolean), or one can use `info methods ?pattern?`, where `pattern` might be a single method name without wild-card characters. The method `info methods ?pattern?` returns a list of matching names, which might be empty. These different methods appear appropriate depending on the context.

XOTcl

```
/obj|cls/ info \
   [inst](commands|procs|parametercmd) \
   ?pattern?
```

Next Scripting Language

```
/cls/ info method exists /methodName/
/cls/ info methods /methodName/
/obj/ info object method exists /methodName/
/obj/ info object methods /methodName/
```

2.6.5. List callable methods

In order to obtain for an object the set of artefacts defined in the class hierarchy, NX uses `info lookup`. One can either lookup methods (via `info lookup methods`) or slots (via `info lookup slots`). The plural term refers to a potential set of return values.

XOTcl

```
/obj/ info methods ?pattern?
```

Next Scripting Language

```
/obj/ info lookup methods ... ?pattern?
# Returns list of method names
```

XOTcl

```
# n.a.
```

Next Scripting Language

```
# List only application specific methods
/obj/ info lookup methods -source application ...
?pattern?
# Returns list of method names
```

XOTcl

```
# Options for 'info methods'
#
# -incontext
# -nomixins
```

Next Scripting Language

```
# Options for 'info lookup methods'
#
# -source ...
# -callprotection ...
# -incontext
# -type ...
# -nomixins
```

XOTcl

```
# n.a.
```

Next Scripting Language

```
# List slot objects defined for obj
# -source might be all|application|baseclasses
# -type is the class of the slot object

/obj/ info lookup slots ?-type ...? ?-source ...?
?pattern?

# Returns list of slot objects
```

XOTcl

```
# List registered filters
/obj/ info filters -order ?-guards? ?pattern?

# List registered mixins
/obj/ info mixins -heritage ?-guards? ?pattern?
```

Next Scripting Language

```
# List registered filters
/obj/ info lookup filters ?-guards? ?pattern?

# List registered mixins
/obj/ info lookup mixins ?-guards? ?pattern?
```

2.6.6. List object/class where a specified method is defined

`info lookup` can be used as well to determine, where exactly an artefact is located. One can obtain

this way a method handle, where a method or filter is defined.

The concept of a *method-handle* is new in NX. The method-handle can be used to obtain more information about the method, such as e.g. the definition of the method.

XOTcl	Next Scripting Language
`/obj/ procsearch /methodName/`	`/obj/ info lookup method /methodName/` `# Returns method-handle`
`/obj/ filtersearch /methodName/`	`/obj/ info lookup filter /methodName/` `# Returns method-handle`

2.6.7. List definition of scripted methods

XOTcl contains a long list of `info` subcommands for different kinds of methods and for obtaining more detailed information about these methods.

In NX, this list of `info` subcommands is much shorter and more orthogonal. For example `info method definition` can be used to obtain with a single command the full definition of a *scripted method*, and furthermore, it works as well the same way to obtain e.g. the definition of a *forwarder* or an *alias*.

While XOTcl uses different names for info options for objects and classes (using the prefix "inst" for instance specific method), NX uses for object specific method the modifier `object`. For definition of class object specific methods, use the modifier `object` as usual.

XOTcl	Next Scripting Language
`# n.a.`	`/cls/ info method definition /methodName/` `/obj/ info object method definition /methodName/`
`/cls/ info instbody /methodName/` `/obj/ info body /methodName/`	`/cls/ info method body /methodName/` `/obj/ info object method body /methodName/`
`/cls/ info instargs /methodName/` `/obj/ info args /methodName/`	`/cls/ info method args /methodName/` `/obj/ info object method args /methodName/`
`/cls/ info instnonposargs /methodName/` `/obj/ info object method args /methodName/`	`/cls/ info method parameter /methodName/` `/obj/ info object method parameter /methodName/`
`/cls/ info instdefault /methodName/` `/obj/ info default /methodName/`	`# not needed, part of` `# "info ?object? method parameter"`
`/cls/ info instpre /methodName/` `/obj/ info pre /methodName/`	`/cls/ info method precondition /methodName/` `/obj/ info object method precondition /methodName/`
`/cls/ info instpost /methodName/` `/obj/ info post /methodName/`	`/cls/ info method postcondition /methodName/` `/obj/ info object method postcondition /methodName/`

Another powerful introspection option in NX is `info ?object? method syntax` which obtains a representation of the parameters of a method in the style of Tcl man pages (regardless of the kind of method).

XOTcl

```
# n.a.
```

Next Scripting Language

```
/cls/ info method syntax /methodName/
/obj/ info object method syntax /methodName/
```

2.6.8. List Configure Parameters

The way, how newly created objects can be configured is determined in NX via properties. The configuration happens during creation via the methods `create` or `new` or during runtime via `configure`. These methods have therefore virtual argument lists, depending on the object or class on which they are applied.

XOTcl

```
# n.a.
```

Next Scripting Language

```
# Return the parameters applicable to
# the create method of a certain class.
# class can be configured. A pattern can
# be used to filter the results.

/cls/ info lookup parameters create ?/pattern/?

# Return in the result in documentation syntax

/cls/ info lookup syntax create ?/pattern/?

# "info lookup parameters configure" returns
# parameters available for configuring the
# current object  (might contain object
# specific information)

/obj/ info lookup parameters configure ?pattern?

# "info lookup configure syntax" returns syntax of
# a call to configure in the Tcl parameter syntax

/obj/ info lookup syntax configure

# Obtain information from a parameter
# (as e.g. returned from "info lookup
# parameters configure").

nsf::parameter::info name /parameter/
nsf::parameter::info syntax /parameter/
nsf::parameter::info type /parameter/
```

2.6.9. List Variable Declarations (property and variable)

XOTcl

```
# obtain parameter definitions defined
# for a class
/cls/ info parameter
```

Next Scripting Language

```
# "info variables" returns handles of
# properties and variables defined by this
# class or object

/cls/ info variables ?pattern?
/obj/ info object variables ?pattern?

# "info lookup variables" returns handles
# of variables and properties applicable
# for the current object (might contain
# object specific information)

/obj/ info lookup variables /pattern/
```

XOTcl

Next Scripting Language

```
# "info variable" lists details about a
# single property or variable.

/obj/ info variable definition /handle/
/obj/ info variable name /handle/
/obj/ info variable parameter /handle/
```

2.6.10. List Slots

XOTcl

```
# n.a.
```

Next Scripting Language

```
# Return list of slots objects defined on the
# object or class
#
# -source might be all|application|baseclasses
# -type is the class of the slot object
# -closure includes slots of superclasses

/cls/ info slots \
   ?-type value? ?-closure? ?-source value?
?pattern?
/obj/ info object slots ?-type ...? ?pattern?

# List reachable slot objects defined for obj
# -source might be all|application|baseclasses
# -type is the class of the slot object
# Returns list of slot objects.

/obj/ info lookup slots \
   ?-type ...? ?-source ... ?pattern?

# Obtain definition, name or parameter from
# slot object

/slotobj/ definition
/slotobj/ name
/slotobj/ parameter
```

2.6.11. List Filter or Mixins

In NX all introspection options for filters are provided via `info filters` and all introspection options for mixins are provided via `info mixins`.

XOTcl

```
/obj/ info filter ?-guards? ?-order? ?pattern?
/obj/ info filterguard /name/
```

Next Scripting Language

```
/obj/ info object filters \
   ?-guards? ?pattern?
```

XOTcl

```
/cls/ info instfilter \
   ?-guards? ?-order? ?pattern?
/cls/ info instfilterguard /name/
```

Next Scripting Language

```
/cls/ info filters \
   ?-guards? ?pattern?
```

XOTcl

```
/obj/ info mixin ?-guards? ?-order ?pattern?
/obj/ info mixinguard /name/
```

Next Scripting Language

```
/obj/ info object mixins \
   ?-guards? ?pattern?
```

XOTcl

```
/cls/ info instmixin \
   ?-guards? ?-order? ?pattern?
/cls/ info instmixinguard /name/
```

Next Scripting Language

```
/cls/ info mixins \
   ?-closure? ?-guards? ?-heritage? ?pattern?
```

2.6.12. List definition of methods defined by aliases, setters or forwarders

As mentioned earlier, `info method definition` can be used on every kind of method. The same call can be used to obtain the definition of a scripted method, a method-alias, a forwarder or a setter method.

XOTcl

```
# n.a.
```

Next Scripting Language

```
/cls/ info method definition /methodName/
/obj/ info object method definition /methodName/
```

2.6.13. List Method-Handles

NX supports *method-handles* to provide means to obtain further information about a method or to change maybe some properties of a method. When a method is created, the method creating method returns the method handle to the created method.

XOTcl

```
# n.a.
```

Next Scripting Language

```
#
# List the method handle of the specified method,
# can be used e.g. for aliases. "handle" is the
short
# form of "definitionhandle".
#
/cls/ info method handle /methodName/
/obj/ info object method handle /methodName/
#
# For ensemble methods (method name contains
# spaces) one can query as well the registration
# handle, which is the handle to the root of the
# ensemble; the definition handle points to the
# leaf of the ensemble.
#
/cls/ info method registrationhandle /methodName/
/obj/ info object method registrationhandle
/methodName/
#
# For aliases, one can query the original
# definition via "info method origin"
#
/cls/ info method origin /methodName/
/obj/ info object method origin /methodName/
```

2.6.14. List type of a method

The method `info ?object? method type` is new in NX to obtain the type of the specified method.

XOTcl

```
# n.a.
```

Next Scripting Language

```
/cls/ info method type /methodName/
/obj/ info object method type /methodName/
```

2.6.15. List the scope of mixin classes

NX provides a richer set of introspection options to obtain information, where mixins classes are mixed into.

XOTcl	Next Scripting Language

`/cls/ info mixinof ?-closure? ?pattern?`	`# List objects, where /cls/ is a`
	`# per-object mixin`
	
	`/cls/ info mixinof -scope object ?-closure? \`
	`  ?pattern?`

`/cls/ info instmixinof ?-closure? ?pattern?`	`# List classes, where /cls/ is a per-class mixin`
	
	`/cls/ info mixinof -scope class ?-closure? \`
	`  ?pattern?`

`# n.a.`	`# List objects and classes, where /cls/ is`
	`# either a per-object or a per-class mixin`
	
	`/cls/ info mixinof -scope all ?-closure? \`
	`  ?pattern?`

	`/cls/ info mixinof ?-closure? ?pattern?`

2.6.16. Check properties of object and classes

Similar as noted before, NX uses rather a hierarchical approach of naming using multiple layers of subcommands).

XOTcl	Next Scripting Language
`/obj/ istype /sometype/`	`# Check if object is a subtype of some class` `/obj/ info has type /sometype/`

`/obj/ ismixin /cls/`	`# Check if object has the specified mixin`
	`registered`
	`/obj/ info has mixin /cls/`

`/obj/ isclass ?/cls/?`	`# Check if object is an NX class`
	`/obj/ has type ::nx::Class`
	
	`# Check if object is a class in one of the`
	`# NSF object systems`
	`::nsf::is class /obj/`

`/obj/ ismetaclass /cls/`	`# Check if class is an NX metaclass`
	`expr {[/cls/ info heritage ::nx::Class] ne ""}`
	
	`# Check if object is a metaclass in one of the`
	`# NSF object systems`
	`::nsf::is metaclass /obj/`

`# n.a.`	`# Check if object is a baseclass of an object`
	`system`
	`::nsf::is baseclass /obj/`

`# n.a.`	`# Return name of object (without namespace prefix)`

XOTcl	Next Scripting Language
	`/obj/ info name`
`/obj/ object::exists /obj/`	`# Check for existence of object (nsf primitive)` `::nsf::object::exists /obj/`

2.6.17. Call-stack Introspection

Call-stack introspection is very similar in NX and XOTcl. NX uses for subcommand the term `current` instead of `self`, since `self` has a strong connotation to the current object. The term `proc` is renamed by `method`.

XOTcl	Next Scripting Language
`self`	`self` `current object`
`self class`	`current class`
`self args`	`current args`
`self proc`	`current method`
`self callingclass`	`current calledclass`
`self callingobject`	`current callingobject`
`self callingproc`	`current callingmethod`
`self calledclass`	`current calledclass`
`self calledproc`	`current calledmethod`
`self isnextcall`	`current isnextcall`

`self next`	`# Returns method-handle of the`
	`# method to be called via "next"`
	`current next`

XOTcl	Next Scripting Language

`self filterreg`	`# Returns method-handle of the`
	`# filter method`
	`current filterreg`

`self callinglevel`	`current callinglevel`
`self activelevel`	`current activelevel`

2.7. Other Predefined Methods

XOTcl	Next Scripting Language
`/obj/ requireNamespace`	`/obj/ require namespace`
`# n.a.`	`/obj/ require method`

2.8. Dispatch, Aliases, etc.

todo: to be done or omitted

2.9. Assertions

In contrary to XOTcl, NX provides no pre-registered methods for assertion handling. All assertion handling can e performed via the Next Scripting primitive `nsf::method::assertion`.

XOTcl	Next Scripting Language
`/obj/ check /checkoptions/`	`::nsf::method::assertion /obj/ check /checkoptions/`
`/obj/ info check`	`::nsf::method::assertion /obj/ check`
`/obj/ invar /conditions/`	`::nsf::method::assertion /obj/ object-invar` `/conditions/`
`/obj/ info invar`	`::nsf::method::assertion /obj/ object-invar`

XOTcl	Next Scripting Language
`/cls/ instinvar /conditions/`	`::nsf::method::assertion /cls/ class-invar /conditions/`
`/cls/ info instinvar`	`::nsf::method::assertion /cls/ class-invar`
`/cls/ invar /conditions/`	`::nsf::method::assertion /cls/ object-invar /conditions/`
`/cls/ info invar`	`::nsf::method::assertion /cls/ object-invar`

2.10. Method Protection

As described above, NX supports method protection via the method modifiers `protected` and `public`. A protected method can be only called from an object of that class, while public methods can be called from every object. The method protection can be used to every kind of method, such as e.g. scripted methods, aliases, forwarders, or accessors. For invocations, the most specific definition (might be a mixin) is used for determining the protection.

3. Incompatibilities between XOTcl 1 and XOTcl 2

3.1. Resolvers

The resolvers (variable resolvers, function resolvers) of the Next Scripting Framework are used as well within XOTcl 2. When variable names or method names starting with a single colon are used in XOTcl 1 scripts, conflicts will arise with the resolver. These names must be replaced.

3.2. Parameters

The following changes for parameters could be regarded as bug-fixes.

3.2.1. Parameter usage without a value

In XOTcl 1, it was possible to call a parameter method during object creation via the dash-interface without a value (in the example below `-x`).

```
# XOTcl example

Class Foo -parameter {x y}
Foo f1 -x -y 1
```

Such cases are most likely mistakes. All parameter configurations in XOTcl 2 require an argument.

3.2.2. Ignored Parameter definitions

In XOTcl 1, a more specific parameter definition without a default was ignored when a more general parameter definition with a default was present. In the example below, the object `b1` contained in XOTcl 1 incorrectly the parameter `x` (set via default from `Foo`), while in XOTcl 2, the variable won't be set.

```
# XOTcl example

Class Foo -parameter {{x 1}}
Class Bar -superclass Foo -parameter x
Bar b1
```

3.2.3. Changing classes and superclasses

NX does not define the methods `class` and `superclass` (like XOTcl), but allows one to alter all object/class relations (including class/superclass/object-mixin/...) `nsf::relation::set`. The class and superclass can be certainly queried in all variants with `info class` or `info superclasses`.

```
# NX example

nx::Class create Foo
Foo create f1

# now alter the class of object f1
nsf::relation::set f1 class ::nx::Object
```

3.2.4. Overwriting procs/methods with objects and vice versa

NSF is now more conservative on object/method creation. In contrary to XOTcl 1 NSF does not allow one to redefined a pre-existing command (e.g. "set") with an object and vice versa. Like in XOTcl 1, preexisting objects and classes con be redefined (necessary for reloading objects/classes in a running interpreter).

3.2.5. Info heritage

`info heritage` returns in XOTcl 1 the transitive superclass hierarchy, which is equivalent with `info superclasses -closure` and therefore not necessary. In XOTcl 2 (and NX), `info heritage` includes as well the transitive per-class mixins.

3.3. Slots

All slot objects (also XOTcl slot objects) are now next-scripting objects of baseclass `::nx::Slot`. The name of the experimental default-setter `initcmd` was changed to `defaultcmd`. Code directly working on the slots objects has to be adapted.

3.4. Obsolete Commands

Parameter-classes were rarely used and have been replaced by the more general object parametrization. Therefore, `cl info parameterclass` has been removed.

3.5. Stronger Checking

The Next Scripting Framework performs stronger checking than XOTcl 1 For example, the requiredness of slots in XOTcl 1 was just a comment, while XOTcl 2 enforces it.

3.6. Exit Handlers

The exit handler interface changed from a method of `::xotcl::Object` into the Tcl command `::nsf::exithandler`:

```
# NX example
::nsf::exithandler set|get|unset ?arg?
```

Version 2.3.0
Last updated 2019-05-07 11:33:18 CEST