Tutorial for the Next Scripting Language

Gustaf Neumann <neumann@wu-wien.ac.at>, Stefan Sobernig <stefan.sobernig@wu.ac.at>

version 2.3.0, May 2019
Written for the Initial Release of the Next Scripting Framework.

Table of Contents

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

Abstract

This document provides a tutorial for the Next Scripting Language NX.

The Next Scripting Language (NX) is a highly flexible object oriented scripting language based on Tcl [Ousterhout 1990]. NX is a successor of XOTcl 1 [Neumann and Zdun 2000a] and was developed based on 10 years of experience with XOTcl in projects containing several hundred 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 is on combining this with *Language Oriented Programming*. In many respects, NX was designed to ease the learning of the language for 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 developers 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 (NSF) 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 name of the Next Scripting Framework is derived from the universal method combinator "next", which was introduced in XOTcl. The combinator "next" serves as a single instrument for method combination with filters, per-object and transitive per-class mixin classes, object methods and multiple inheritance.

The 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.

1. NX and its Roots

Object oriented extensions of Tcl have quite a long history. Two of the most prominent early Tcl based OO languages were *incr Tcl* (abbreviated as itcl) and Object Tcl (*OTcl* [Wetherall and Lindblad 1995]). While itcl provides a traditional C++/Java-like object system, OTcl was following the CLOS approach and supports a dynamic object system, allowing incremental class and object extensions and re-classing of objects.

Extended Object Tcl (abbreviated as XOTcl [Neumann and Zdun 2000a]) is a successor of OTcl and was the first language providing language support for design patterns. XOTcl extends OTcl by providing namespace support, adding assertions, dynamic object aggregations, slots and by introducing per-object and per-class filters and per-object and per-class mixins.

XOTcl was so far released in more than 30 versions. It is described in its detail in more than 20 papers and serves as a basis for other object systems like TclOO [Donal ???]. The scripting language *NX* and the *Next Scripting Framework* [Neumann and Sobernig 2009] extend the basic ideas of XOTcl by providing support for *language-oriented programming*. The 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 can follow different naming conventions for built-in methods. Currently, the Next Scripting Framework is packaged with three object systems: NX, XOTcl 2.0, and TclCool (the language introduced by TIP#279).

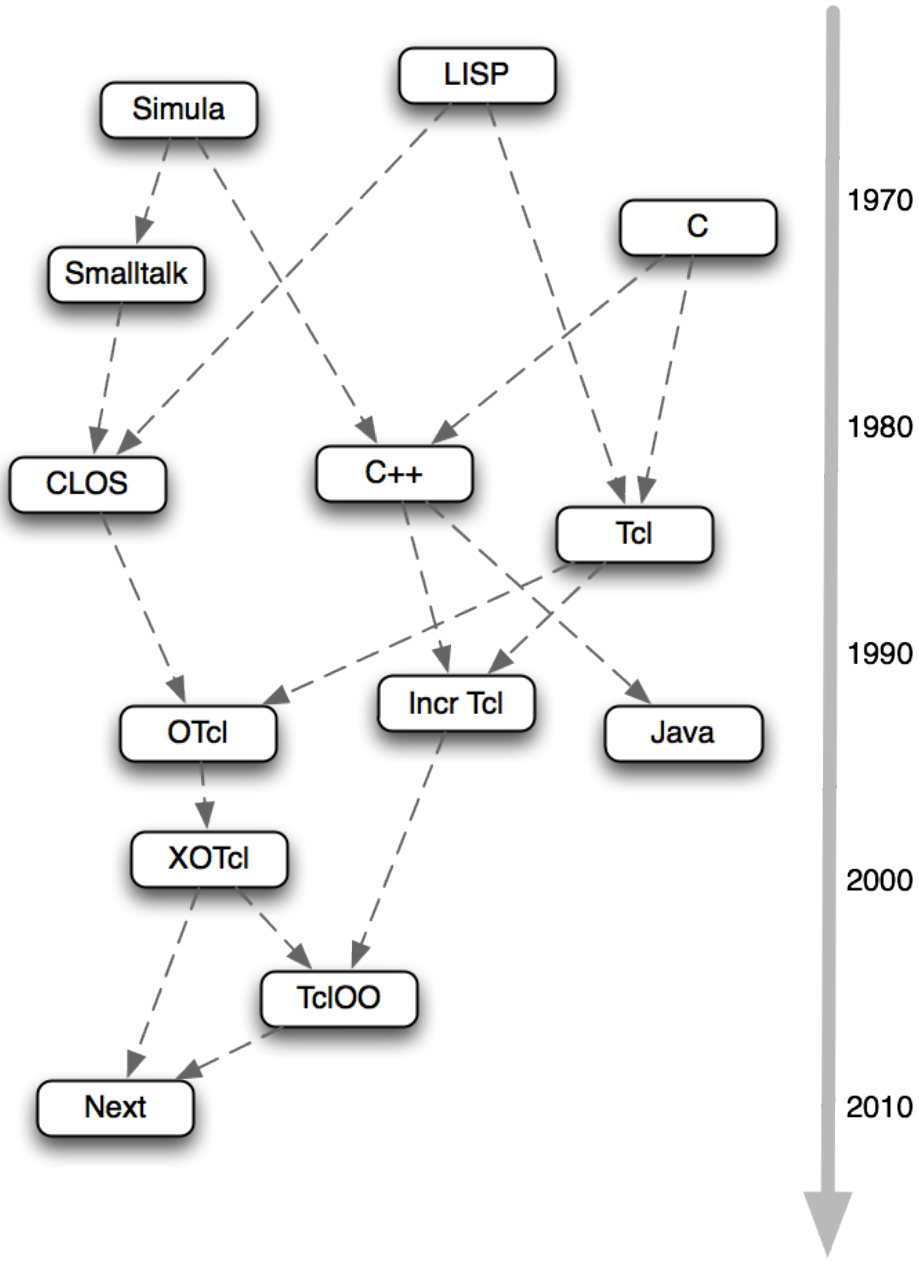

Figure 1. Language History of the Next Scripting Language

The primary purpose of this document is to introduce NX to beginners. We expect some prior knowledge of programming languages, and some knowledge about Tcl. In the following sections we introduce NX by examples. In later sections we introduce the more advanced concepts of the language. Conceptually, most of the addressed concepts are very similar to XOTcl. Concerning the differences between NX and XOTcl, please refer to the *Migration Guide for the Next Scripting Language*.

2. Introductory Overview Example: Stack

A classical programming example is the implementation of a stack, which is most likely familiar to many readers from many introductory programming courses. A stack is a last-in first-out data structure which

is manipulated via operations like `push` (add something to the stack) and `pop` remove an entry from the stack. These operations are called *methods* in the context of object oriented programming systems. Primary goals of object orientation are encapsulation and abstraction. Therefore, we define a common unit (a class) that defines and encapsulates the behavior of a stack and provides methods to a user of the data structure that abstract from the actual implementation.

2.1. Define a Class "Stack"

In our first example, we define a class named `Stack` with the methods `push` and `pop`. When an instance of the stack is created (e.g. a concrete stack `s1`) the stack will contain an instance variable named `things`, initialized with the an empty list.

Listing 2: Class Stack

```
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
   }
}
```

Typically, classes are defined in NX via `nx::Class create`, followed by the name of the new class (here: `Stack`). The definition of the stack placed between curly braces and contains here just the method definitions. Methods of the class are defined via `:method` followed by the name of the method, an argument list and the body of the method, consisting of Tcl and NX statements.

When an instance of `Stack` is created, it will contain an instance variable named `things`. If several `Stack` instances are created, each of the instances will have their own (same-named but different) instance variable. The instance variable `things` is used in our example as a list for the internal representation of the stack. We define in a next step the methods to access and modify this list structure. A user of the stack using the provided methods does not have to have any knowledge about the name or the structure of the internal representation (the instance variable `things`).

The method `push` receives an argument `thing` which should be placed on the stack. Note that we do not have to specify the type of the element on the stack, so we can push strings as well as numbers or other kind of things. When an element is pushed, we add this element as the first element to the list `things`. We insert the element using the Tcl command `linsert` which receives the list as first element, the position where the element should be added as second and the new element as third argument. To access the value of the instance variable we use Tcl's dollar operator followed by the name. The names of instance variables are preceded with a colon `:`. Since the name contains a non-plain character, Tcl requires us to put braces around the name. The command `linsert` and its arguments are placed between square brackets. This means that the function `linsert` is called and a new list is returned, where the new element is inserted at the first position (index 0) in the list `things`. The result of the `linsert` function is assigned again to the instance variable `things`, which is updated this way. Finally the method `push` returns the pushed thing using the `return` statement.

The method `pop` returns the most recently stacked element and removes it from the stack. Therefore, it takes the first element from the list (using the Tcl command `lindex`), assigns it to the method-scoped variable `top`, removes the element from the instance variable `things` (by using the Tcl command `lrange`) and returns the value popped element `top`.

This finishes our first implementation of the stack, more enhanced versions will follow. Note that the methods `push` and `pop` are defined as `public`; this means that these methods can be used from all other objects in the system. Therefore, these methods provide an interface to the stack implementation.

Listing 3: Using the Stack

```
#!/usr/bin/env tclsh
package require nx

nx::Class create Stack {

   #
   # Stack of Things
   #
   ....
}

Stack create s1
s1 push a
s1 push b
s1 push c
puts [s1 pop]
puts [s1 pop]
s1 destroy
```

Now we want to use the stack. The code snippet in Listing 3 shows how to use the class Stack in a script. Since NX is based on Tcl, the script will be called with the Tcl shell `tclsh`. In the Tcl shell we have to `require package nx` to use the Next Scripting Framework and NX. The next lines contain the definition of the stack as presented before. Of course, it is as well possible to make the definition of the stack an own package, such we could simple say `package require stack`, or to save the definition of a stack simply in a file and load it via `source`.

In line 12 we create an instance of the stack, namely the stack object `s1`. The object `s1` is an instance of `Stack` and has therefore access to its methods. The methods like `push` or `pop` can be invoked via a command starting with the object name followed by the method name. In lines 13-15 we push on the stack the values `a`, then `b`, and `c`. In line 16 we output the result of the `pop` method using the Tcl command `puts`. We will see on standard output the value+c+ (the last stacked item). The output of the line 17 is the value `b` (the previously stacked item). Finally, in line 18 we destroy the object. This is not necessary here, but shows the life cycle of an object. In some respects, `destroy` is the counterpart of `create` from line 12.

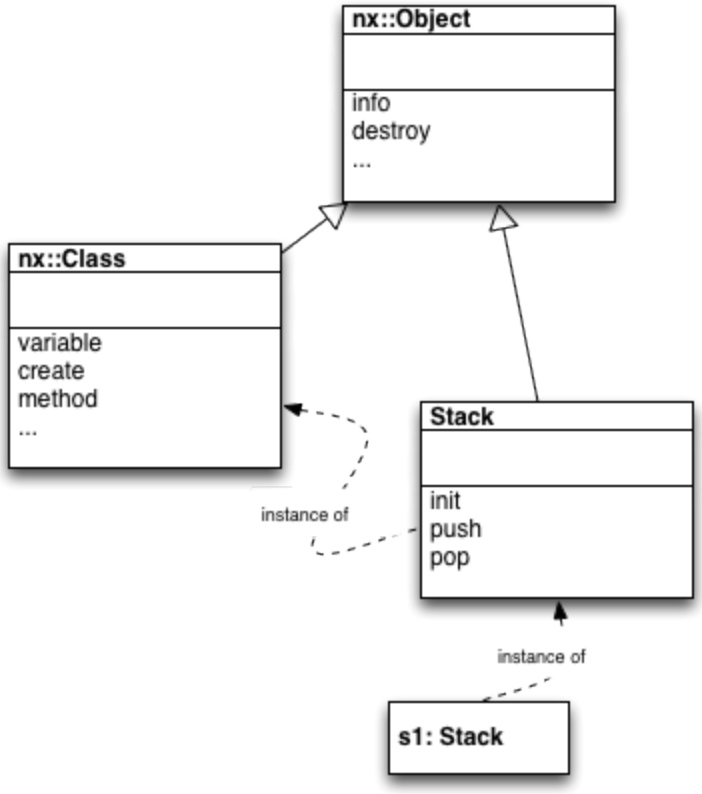

Figure 4. Class and Object Diagram

Figure 4 shows the actual class and object structure of the first `Stack` example. Note that the common root class is `nx::Object` that contains methods for all objects. Since classes are as well objects in NX, `nx::Class` is a specialization of `nx::Object`. `nx::Class` provides methods for creating objects, such as the method `create` which is used to create objects (and classes as well).

2.2. Define an Object Named "stack"

The definition of the stack in Listing 2 follows the traditional object oriented approach, found in practically every object oriented programming language: Define a class with some methods, create instances from this class, and use the methods defined in the class in the instances of the class.

In our next example, we introduce *generic objects* and *object specific methods*. With NX, we can define generic objects, which are instances of the most generic class `nx::Object` (sometimes called *common root class*). `nx::Object` is predefined and contains a minimal set of methods applicable to all NX objects. In this example, we define a generic object named `stack` and provide methods for this object. The methods defined above were methods provided by a class for objects. Now we define object specific methods, which are methods applicable only to the object for which they are defined.

Listing 5: Object stack

```
nx::Object create stack {

   :object variable things {}

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

```
    }

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

The example in Listing 5 defines the object `stack` in a very similar way as the class `Stack`. But the following points are different.

- First, we use `nx::Object` instead of `nx::Class` to denote that we want to create a generic object, not a class.
- We use `:object variable` to define the variable `things` just for this single instance (the object `stack`).
- The definition for the methods `push` and `pop` are the same as before, but here we defined these with `object method`. Therefore, these two methods `push` and `pop` are object-specific.

In order to use the stack, we can use directly the object `stack` in the same way as we have used the object `s1` in Listing 3 the class diagram for this the object `stack`.

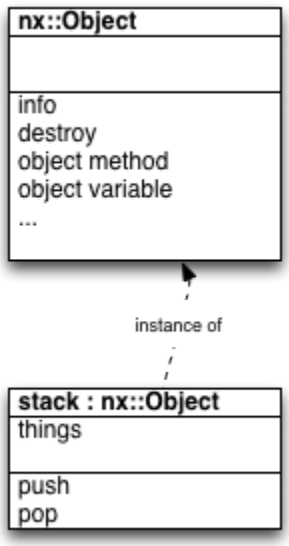

Figure 6. Object stack

A reader might wonder when to use a class `Stack` or rather an object `stack`. A big difference is certainly that one can define easily multiple instances of a class, while the object is actually a single, tailored entity. The concept of the object `stack` is similar to a module, providing a certain functionality via a common interface, without providing the functionality to create multiple instances. The reuse of methods provided by the class to objects is as well a difference. If the methods of the class are updated, all instances of the class will immediately get the modified behavior. However, this does not mean that the reuse for the methods of `stack` is not possible. NX allows for example to copy objects (similar to prototype based languages) or to reuse methods via e.g. aliases (more about this later).

Note that we use capitalized names for classes and lowercase names for instances. This is not required and a pure convention making it easier to understand scripts without much analysis.

2.3. Implementing Features using Mixin Classes

So far, the definition of the stack methods was pretty minimal. Suppose, we want to define "safe stacks" that protect e.g. against stack under-runs (a stack under-run happens, when more `pop` than `push` operations are issued on a stack). Safety checking can be implemented mostly independent from the implementation details of the stack (usage of internal data structures). There are as well different ways of checking the safety. Therefore we say that safety checking is orthogonal to the stack core implementation.

With NX we can define stack-safety as a separate class using methods with the same names as the implementations before, and "mix" this behavior into classes or objects. The implementation of `Safety` in stack under-runs and to issue error messages, when this happens.

Listing 7: Class Safety

```
nx::Class create Safety {

  #
  # Implement stack safety by defining an additional
  # instance variable named "count" that keeps track of
  # the number of stacked elements. The methods of
  # this class have the same names and argument lists
  # as the methods of Stack; these methods "shadow"
  # the methods of class Stack.
  #

  :variable count 0

  :public method push {thing} {
    incr :count
    next
  }

  :public method pop {} {
    if {${:count} == 0} { error "Stack empty!" }
    incr :count -1
    next
  }
}
```

Note that all the methods of the class `Safety` end with `next`. This command is a primitive command of NX, which calls the same-named method with the same argument list as the current invocation.

Assume we save the definition of the class `Stack` in a file named `Stack.tcl` and the definition of the class `Safety` in a file named `Safety.tcl` in the current directory. When we load the classes `Stack` and `Safety` into the same script (see the terminal dialog in e.g. a certain stack `s2` as a safe stack, while all other stacks (such as `s1`) might be still "unsafe". This can be achieved via the option `-mixin` at the object creation time (see line 9 in option `-mixin` mixes the class `Safety` into the new instance `s2`.

Listing 8: Using the Class Safety

```
% package require nx
2.0
% source Stack.tcl
::Stack
% source Safety.tcl
::Safety
% Stack create s1
::s1
% Stack create s2 -object-mixin Safety
::s2
```

```
% s2 push a
% s2 pop
a
% s2 pop
Stack empty!

% s1 info precedence
::Stack ::nx::Object

% s2 info precedence
::Safety ::Stack ::nx::Object
```

When the method `push` of `s2` is called, first the method of the mixin class `Safety` will be invoked that increments the counter and continues with `next` to call the shadowed method, here the method `push` of the `Stack` implementation that actually pushes the item. The same happens, when `s2 pop` is invoked, first the method of `Safety` is called, then the method of the `Stack`. When the stack is empty (the value of `count` reaches 0), and `pop` is invoked, the mixin class `Safety` generates an error message (raises an exception), and does not invoke the method of the `Stack`.

The last two commands in Listing 8 use introspection to query for the objects `s1` and `s2` in which order the involved classes are processed. This order is called the `precedence order` and is obtained via `info precedence`. We see that the mixin class `Safety` is only in use for `s2`, and takes there precedence over `Stack`. The common root class `nx::Object` is for both `s1` and `s2` the base class.

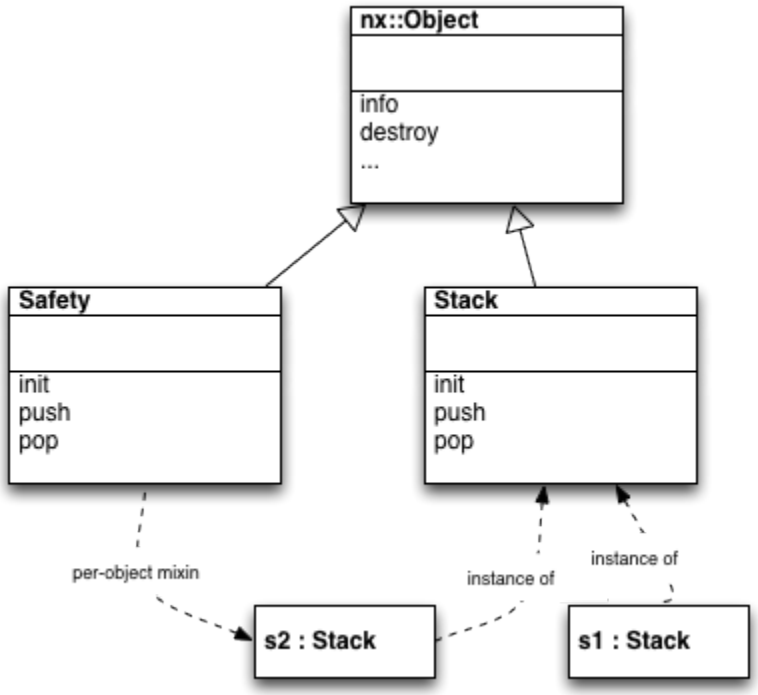

Figure 9. Per-object Mixin

Note that in Listing 8, the class `Safety` is only mixed into a single object (here `s2`), therefore we refer to this case as a *per-object mixin*. Figure 9 shows the class diagram, where the class `Safety` is used as a per-object mixin for `s2`.

The mixin class `Safety` can be used as well in other ways, such as e.g. for defining classes of safe stacks:

Listing 10: Class SafeStack

```
#
# Create a safe stack class by using Stack and mixin
# Safety
#
nx::Class create SafeStack -superclasses Stack -mixins Safety

SafeStack create s3
```

The difference of a per-class mixin and a per-object mixin is that the per-class mixin is applicable to all instances of the class. Therefore, we call these mixins also sometimes instance mixins. In our example in Listing 10, `Safety` is mixed into the definition of `SafeStack`. Therefore, all instances of the class `SafeStack` (here the instance `s3`) will be using the safety definitions.

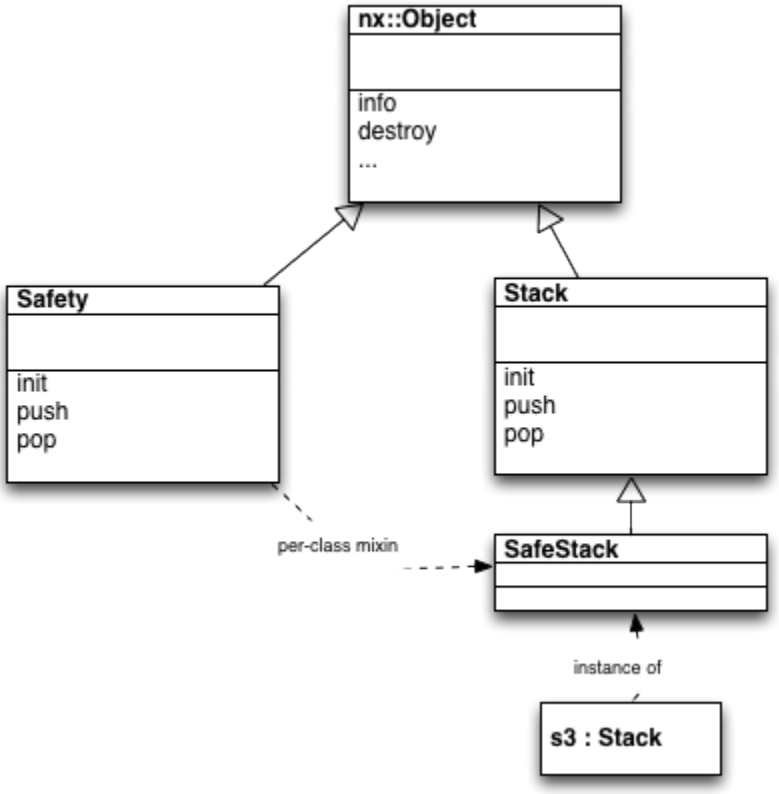

Figure 11. Per-class Mixin

Figure 11 shows the class diagram for this definition. Note that we could use `Safety` as well as a per-class mixin on `Stack`. In this case, all stacks would be safe stacks and we could not provide a selective feature selection (which might be perfectly fine).

2.4. Define Different Kinds of Stacks

The definition of `Stack` is generic and allows all kind of elements to be stacked. Suppose, we want to use the generic stack definition, but a certain stack (say, stack `s4`) should be a stack for integers only. This behavior can be achieved by the same means as introduced already in Listing 5, namely object-specific methods.

Listing 12: Object Integer Stack

```
Stack create s4 {

  #
  # Create a stack with a object-specific method
  # to check the type of entries
  #

  :public object method push {thing:integer} {
    next
  }
}
```

The program snippet in Listing 12 defines an instance `s4` of the class `Stack` and provides an object specific method for `push` to implement an integer stack. The method `pull` is the same for the integer stack as for all other stacks, so it will be reused as usual from the class `Stack`. The object-specific method `push` of `s4` has a value constraint in its argument list (`thing:integer`) that makes sure that only integers can be stacked. In case the argument is not an integer, an exception will be raised. Of course, one could perform the value constraint checking as well in the body of the method `proc` by accepting an generic argument and by performing the test for the value in the body of the method. In the case, the passed value is an integer, the `push` method of Listing 12 calls `next`, and therefore calls the shadowed generic definition of `push` as provided by `Stack`.

Listing 13: Class IntegerStack

```
nx::Class create IntegerStack -superclass Stack {

  #
  # Create a Stack accepting only integers
  #

  :public method push {thing:integer} {
    next
  }
}
```

An alternative approach is shown in Listing 13, where the class `IntegerStack` is defined, using the same method definition as `s4`, this time on the class level.

2.5. Define Object Specific Methods on Classes

In our previous examples we defined methods provided by classes (applicable for their instances) and object-specific methods (methods defined on objects, which are only applicable for these objects). In this section, we introduce methods that are defined on the class objects. Such methods are sometimes called *class methods* or *static methods*.

In NX classes are objects, they are specialized objects with additional methods. Methods for classes are often used for managing the life-cycles of the instances of the classes (we will come to this point in later sections in more detail). Since classes are objects, we can use exactly the same notation as above to define class methods by using `object method`. The methods defined on the class object are in all respects identical with object specific methods shown in the examples above.

Listing 14: Class Stack2

```
nx::Class create Stack2 {

   :public object method available_stacks {} {
      return [llength [:info instances]]
   }
```

```
    :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
    }
}

Stack2 create s1
Stack2 create s2

puts [Stack2 available_stacks]
```

The class `Stack2` in Listing 14 consists of the earlier definition of the class `Stack` and is extended by the class-specific method `available_stacks`, which returns the current number of instances of the stack. The final command `puts` (line 26) prints 2 to the console.

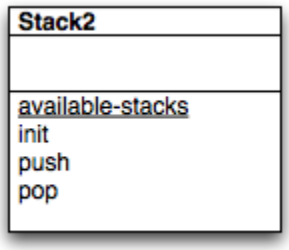

Figure 15. Stack2

The class diagram in Figure 15 shows the diagrammatic representation of the class object-specific method `available_stacks`. Since every class is a specialization of the common root class `nx::Object`, the common root class is often omitted from the class diagrams, so it was omitted here as well in the diagram.

3. Basic Language Features of NX

3.1. Variables and Properties

In general, NX does not need variable declarations. It allows one to create or modify variables on the fly by using for example the Tcl commands `set` and `unset`. Depending on the variable name (or more precisely, depending on the variable name's prefix consisting of colons "`:`") a variable is either local to a method, or it is an instance variable, or a global variable. The rules are:

- A variable without any colon prefix refers typically to a method scoped variable. Such a variable is created during the invocation of the method, and it is deleted, when the method ends. In the example below, the variable `a` is method scoped.
- A variable with a single colon prefix refers to an instance variable. An instance variable is part of the object; when the object is destroyed, its instance variables are deleted as well. In the example below, the variable `b` is an instance variable.

- A variable with two leading colons refers to a global variable. The lifespan of a globale variable ends when the variable is explicitly unset or the script terminates. Variables, which are placed in Tcl namespaces, are also global variables. In the example below, the variable `c` is a global variable.

Listing 16: Scopes of Variables

```
nx::Class create Foo {

  :public method foo args {...}
    # "a" is a method scoped variable
    set a 1
    # "b" is an Instance variable
    set :b 2
    # "c" is a global variable/namespaced variable
    set ::c 3
  }
}
```

Listing 16 shows a method `foo` of some class `Foo` referring to differently scoped variables.

3.1.1. Properties: Configurable Instance Variables

As described above, there is no need to declare instance variables in NX. In many cases, a developer might want to define some value constraints for variables, or to provide defaults, or to make variables configurable upon object creation. Often, variables are "inherited", meaning that the variables declared in a general class are also available in a more specialized class. For these purposes NX provides *variable handlers* responsible for the management of instance variables. We distinguish in NX between configurable variables (called `property`) and variables that are not configurable (called `variable`).

> A **property** is a definition of a configurable instance variable.

The term configurable means that (a) one can provide at creation time of an instance a value for this variable, and (b), one can query the value via the accessor function `cget` and (c), one can change the value of the variable via `configure` at runtime. Since the general accessor function `cget` and `configure` are available, an application developer does not have to program own accessor methods. When value checkers are provided, each time, the value of the variable is to be changed, the constrained are checked as well.

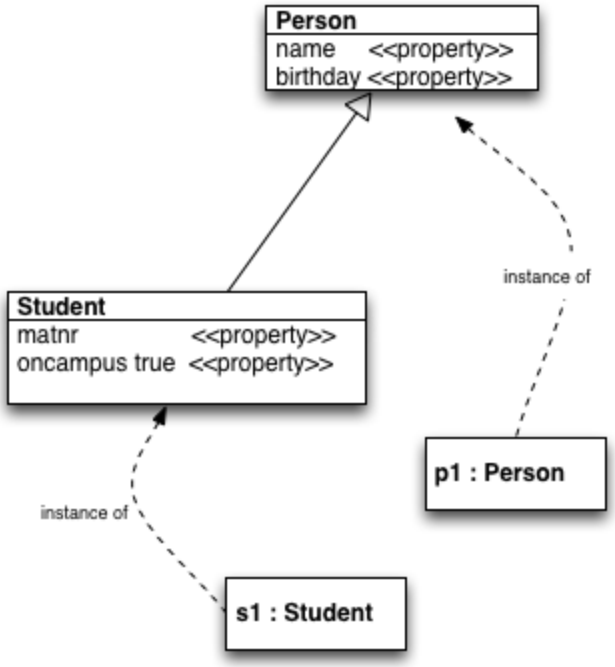

Figure 17. Classes Person and Student

The class diagram above defines the classes `Person` and `Student`. For both classes, configurable instance variable are specified by defining these as properties. The listing below shows an implementation of this conceptual model in NX.

Listing 18: Properties

```
#
# Define a class Person with properties "name"
# and "birthday"
#
nx::Class create Person {
  :property name:required
  :property birthday
}

#
# Define a class Student as specialization of Person
# with additional properties
#
nx::Class create Student -superclass Person {
  :property matnr:required
  :property {oncampus:boolean true}
}

#
# Create instances using configure parameters
# for the initialization
#
Person create p1 -name Bob
Student create s1 -name Susan -matnr 4711

# Access property value via accessor method
puts "The name of s1 is [s1 cget -name]"
```

By defining `name` and `birthday` as properties of `Person`, NX makes these configurable. When we create an instance of `Person` named `p1`, we can provide a value for e.g. the name by specifying `-name`

during creation. The properties result in non-positional configure parameters which can be provided in any order. In our listing, we create an instance of `Person` using the configure parameter `name` and provide the value of `Bob` to the instance variable `name`.

The class `Student` is defined as a specialization of `Person` with two additional properties: `matnr` and `oncampus`. The property `matnr` is required (it has to be provided, when an instance of this class is created), and the property `oncampus` is boolean, and is per default set to `true`. Note that the class `Student` inherits the properties of `Person`. So, `Student` has four properties in total.

The property definitions provide the `configure parameters` for instance creation. Many other languages require such parameters to be passed via arguments of a constructor, which is often error prone, when values are to be passed to superclasses. Also in dynamic languages, the relationships between classes can be easily changed, and different superclasses might have different requirements in their constructors. The declarative approach in NX reduces the need for tailored constructor methods significantly.

Note that the property `matnr` of class `Student` is required. This means, that if we try to create an instance of `Student`, a runtime exception will be triggered. The property `oncamups` is boolean and contains a default value. Providing a default value means that whenever we create an instance of this class the object will contain such an instance variable, even when we provide no value via the configure parameters.

In our listing, we create an instance of `Student` using the two configure parameters `name` and `matnr`. Finally, we use method `cget` to obtain the value of the instance variable `name` of object `s1`.

3.1.2. Non-configurable Instance Variables

In practice, not all instance variables should be configurable. But still, we want to be able to provide defaults similar to properties. To define non-configurable instance variables the predefined method `variable` can be used. Such instance variables are often used for e.g. keeping the internal state of an object. The usage of `variable` is in many respects similar to `property`. One difference is, that `property` uses the same syntax as for method parameters, whereas `variable` receives the default value as a separate argument (similar to the `variable` command in plain Tcl). The introductory Stack example in Listing 2 uses already the method `variable`.

Listing 19: Declaring Variables

```
nx::Class create Base {
  :variable x 1
  # ...
}

nx::Class create Derived -superclass Base {
  :variable y 2
  # ...
}

# Create instance of the class Derived
Derived create d1

# Object d1 has instance variables
# x == 1 and y == 2
```

Note that the variable definitions are inherited in the same way as properties. The example in Listing 19 shows a class `Derived` that inherits from `Base`. When an instance `d1` is created, it will contain the two instance variables `x` and `y`. Note that the variable declarations from `property` and `variable` are used to initialize (and to configure) the instances variables of an object.

Listing 20: Setting Variables in the Constructor

```
nx::Class create Base2 {
 # ...
 :method init {} {
   set :x 1
   # ....
 }
}

nx::Class create Derived2 -superclass Base2 {
 # ...
 :method init {} {
   set :y 2
   next
   # ....
 }
}

# Create instance of the class Derived2
Derived2 create d2
```

In many other object oriented languages, the instance variables are initialized solely by the constructor (similar to class `Derived2` in Listing 20). This approach is certainly also possible in NX. Note that the approach using constructors requires an explicit method chaining between the constructors and is less declarative than the approach in NX using `property` and `variable`.

Both, `property` and `variable` provide much more functionalities. One can for example declare `public`, `protected` or `private` accessor methods, or one can define variables to be incremental (for e.g. adding values to a list of values), or one can define variables specific behavior.

3.2. Method Definitions

The basic building blocks of an object oriented program are object and classes, which contain named pieces of code, the methods.

> **Methods** are subroutines (pieces of code) associated with objects and/or classes. A method has a name, receives optionally arguments during invocation and returns a value.

Plain Tcl provides subroutines, which are not associated with objects or classes. Tcl distinguishes between +proc+s (scripted subroutines) and commands (system-languages implemented subroutines).

Methods might have different scopes, defining, on which kind of objects these methods are applicable to. These are described in more detail later on. For the time being, we deal here with methods defined on classes, which are applicable for the instance of these classes.

3.2.1. Scripted Methods

Since NX is a scripting language, most methods are most likely scripted methods, in which the method body contains Tcl code.

Listing 21: Scripted method

```
# Define a class
nx::Class create Dog {

  # Define a scripted method for the class
  :public method bark {} {
    puts "[self] Bark, bark, bark."
  }
}

# Create an instance of the class
```

```
Dog create fido

# The following line prints "::fido Bark, bark, bark."
fido bark
```

In the example above we create a class `Dog` with a scripted method named `bark`. The method body defines the code, which is executed when the method is invoked. In this example, the method `bar` prints out a line on the terminal starting with the object name (this is determined by the built in command `self`) followed by "Bark, bark, bark.". This method is defined on a class and applicable to instances of the class (here the instance `fido`).

3.2.2. C-implemented Methods

Not all of the methods usable in NX are scripted methods; many predefined methods are defined in the underlying system language, which is typically C. For example, in Listing 21 we used the method `create` to create the class `Dog` and to create the dog instance `fido`. These methods are implemented in C in the next scripting framework.

C-implemented methods are not only provided by the underlying framework but might be as well defined by application developers. This is an advanced topic, not covered here. However, application developer might reuse some generic C code to define their own C-implemented methods. Such methods are for example *accessors*, *forwarders* and *aliases*.

> An **accessor method** is a method that accesses instance variables of an object. A call to an accessor without arguments uses the accessor as a getter, obtaining the actual value of the associated variable. A call to an accessor with an argument uses it as a setter, setting the value of the associated variable.

NX provides support for C-implemented accessor methods. Accessors have already been mentioned in the section about properties. When the option `-accessor public|protected|private` is provided to a `variable` or `property` definition, NX creates automatically a same-named accessors method.

Listing 22: Accessor Methods

```
nx::Class create Dog {
 :public method bark {} { puts "[self] Bark, bark, bark." }
 :method init {} { Tail create [self]::tail}
}

nx::Class create Tail {
  :property -accessor public {length:double 5}
  :public method wag {} {return Joy}
}

# Create an instance of the class
Dog create fido

# Use the accessor "length" as a getter, to obtain the value
# of a property. The following call returns the length of the
# tail of fido
fido::tail length get

# Use the accessor "length" as a setter, to alter the value
# of a property. The following call changes the length of
# the tail of fido
fido::tail length set 10

# Proving an invalid values will raise an error
fido::tail length set "Hello"
```

Listing 22 shows an extended example, where every dog has a tail. The object `tail` is created as a

subobject of the dog in the constructor `init`. The subobject can be accessed by providing the full name of the subobject `fido::tail`. The method `length` is an C-implemented accessor, that enforces the value constraint (here a floating point number, since length uses the value constraint `double`). Line 25 will therefore raise an exception, since the provided values cannot be converted to a double number.

Listing 23: Forwarder Methods

```
nx::Class create Dog {
  :public method bark {} { puts "[self] Bark, bark, bark." }
  :method init {} {
    Tail create [self]::tail
    :public object forward wag [self]::tail wag
  }
}

nx::Class create Tail {
  :property {length 5}
  :public method wag {} {return Joy}
}

# Create an instance of the class
Dog create fido

# The invocation of "fido wag" is delegated to "fido::tail wag".
# Therefore, the following method returns "Joy".
fido wag
```

Listing 23 again extends the example by adding a forwarder named `wag` to the object (e.g. `fido`). The forwarder redirects all calls of the form `fido wag` with arbitrary arguments to the subobject `fido::tail`.

> A **forwarder method** is a C-implemented method that redirects an invocation for a certain method to either a method of another object or to some other method of the same object. Forwarding an invocation of a method to some other object is a means of delegation.

The functionality of the forwarder can just as well be implemented as a scripted method, but for the most common cases, the forward implementation is more efficient, and the `forward` method expresses the intention of the developer.

The method `forwarder` has several options to change e.g. the order of the arguments, or to substitute certain patterns in the argument list etc. This will be described in later sections.

3.2.3. Method-Aliases

> An **alias method** is a means to register either an existing method, or a Tcl proc, or a Tcl command as a method with the provided name on a class or object.

In some way, the method alias is a restricted form of a forwarder, though it does not support delegation to different objects or argument reordering. The advantage of the method alias compared to a forwarder is that it has close to zero overhead, especially for aliasing c-implemented methods.

Listing 24: Method-Alias

```
nx::Class create Dog {
  :public method bark {} { puts "[self] Bark, bark, bark." }

  # Define a public alias for the method "bark"
  :public alias warn [:info method handle bark]
  # ...
}
```

```
# Create an instance of the class
Dog create fido

# The following line prints "::fido Bark, bark, bark."
fido warn
```

Listing 24 extends the last example by defining an alias for the method `bark`. The example only shows the bare mechanism. In general, method aliases are very powerful means for reusing pre-existing functionality. The full object system of NX and XOTcl2 is built from aliases, reusing functionality provided by the next scripting framework under different names. Method aliases are as well a means for implementing traits in NX.

3.3. Method Protection

All kinds of methods might have different kind of protections in NX. The call-protection defines from which calling context methods might be called. The Next Scripting Framework supports as well redefinition protection for methods.

NX distinguishes between `public`, `protected` and `private` methods, where the default call-protection is `protected`.

> A **public** method can be called from every context. A **protected** method can only be invoked from the same object. A **private** method can only be invoked from methods defined on the same entity (defined on the same class or on the same object) via the invocation with the local flag (i.e. "`: -local foo`").

All kind of method protections are applicable for all kind of methods, either scripted or C-implemented.

The distinction between public and protected leads to interfaces for classes and objects. Public methods are intended for consumers of these entities. Public methods define the intended ways of providing methods for external usages (usages, from other objects or classes). Protected methods are intended for the implementor of the class or subclasses and not for public usage. The distinction between protected and public reduces the coupling between consumers and the implementation, and offers more flexibility to the developer.

Listing 25: Protected Methods

```
nx::Class create Foo {

  # Define a public method
  :public method foo {} {
    # ....
    return [:helper]
  }

  # Define a protected method
  :method helper {} {
     return 1
  }
}

# Create an instance of the class:
Foo create f1

# The invocation of the public method "foo" returns 1
f1 foo

# The invocation of the protected method "helper" raises an error:
f1 helper
```

The example above uses `:protected method helper ...`. We could have used here as well `:method helper ...`, since the default method call-protection is already protected.

The method call-protection of `private` goes one step further and helps to hide implementation details also for implementors of subclasses. Private methods are a means for avoiding unanticipated name clashes. Consider the following example:

Listing 26: Private Methods

```
nx::Class create Base {
  :private method helper {a b} {expr {$a + $b}}
  :public method foo     {a b} {: -local helper $a $b}
}

nx::Class create Sub -superclass Base {
  :public method bar     {a b} {: -local helper $a $b}
  :private method helper {a b} {expr {$a * $b}}
  :create s1
}

s1 foo 3 4     ;# returns 7
s1 bar 3 4     ;# returns 12
s1 helper 3 4  ;# raises error: unable to dispatch method helper
```

The base class implements a public method `foo` using the helper method named `helper`. The derived class implements a as well a public method `bar`, which is also using a helper method named `helper`. When an instance `s1` is created from the derived class, the method `foo` is invoked which uses in turn the private method of the base class. Therefore, the invocation `s1 foo 3 4` returns its sum. If the `local` flag had not beed used in helper, `s1` would have tried to call the helper of `Sub`, which would be incorrect. For all other purposes, the private methods are "invisible" in all situations, e.g., when mixins are used, or within the `next`-path, etc.

By using the `-local` flag at the call site it is possible to invoke only the local definition of the method. If we would call the method without this flag, the resolution order would be the standard resolution order, starting with filters, mixins, object methods and the full intrinsic class hierarchy.

NX supports the modifier `private` for methods and properties. In all cases `private` is an instrument to avoid unanticipated interactions and means actually "accessible for methods defined on the same entity (object or class)". The main usage for `private` is to improve locality of the code e.g. for compositional operations.

In order to improve locality for properties, a private property defines therefore internally a variable with a different name to avoid unintended interactions. The variable should be accessed via the private accessor, which can be invoked with the `-local` flag. In the following example class `D` introduces a private property with the same name as a property in the superclass.

Listing 27: Private Properties

```
#
# Define a class C with a property "x" and a public accessor
#
nx::Class create C {
  :property -accessor public {x c}
}

#
# Define a subclass D with a private property "x"
# and a method bar, which is capable of accessing
# the private property.
#
nx::Class create D -superclass C {
  :property -accessor private {x d}
  :public method bar {p} {return [: -local $p get]}
```

```
}

#
# The private and public (or protected) properties
# define internally separate variable that do not
# conflict.
#
D create d1
puts [d1 x get]   ;# prints "c"
puts [d1 bar x]   ;# prints "d"
```

Without the `private` definition of the property, the definition of property `x` in class `D` would shadow the definition of the property in the superclass `C` for its instances (`d1 x` or `set :x` would return `d` instead of `c`).

3.4. Applicability of Methods

As defined above, a method is a subroutine defined on an object or class. This object (or class) contains the method. If the object (or class) is deleted, the contained methods will be deleted as well.

3.4.1. Instance Methods

> Typically, methods are defined on a class, and the methods defined on the class are applicable to the instances (direct or indirect) of this class. These methods are called **instance methods**.

In the following example method, `foo` is an instance method defined on class `C`.

Listing 28: Methods applicable for instances

```
nx::Class create C {
  :public method foo {} {return 1}
  :create c1
}

# Method "foo" is defined on class "C"
# and applicable to the instances of "C"
c1 foo
```

There are many programming languages that only allow these types of methods. However, NX also allows methods to be defined on objects.

3.4.2. Object Methods

> Methods defined on objects are **object methods**. Object methods are only applicable on the object, on which they are defined. Object methods cannot be inherited from other objects.

The following example defines an object method `bar` on the instance `c1` of class `C`, and as well as the object specific method `baz` defined on the object `o1`. An object method is defined via `object method`.

Note that we can define a object method that shadows (redefines) for this object methods provided from classes.

Listing 29: Object Method

```
nx::Class create C {
  :public method foo {} {return 1}
  :create c1 {
```

```
      :public object method foo {} {return 2}
      :public object method bar {} {return 3}
  }
}

# Method "bar" is an object specific method of "c1"
c1 bar

# object-specific method "foo" returns 2
c1 foo

# Method "baz" is an object specific method of "o1"
nx::Object create o1 {
  :public object method baz {} {return 4}
}
o1 baz
```

3.4.3. Class Methods

> A **class method** is a method defined on a class, which is only applicable to the class object itself. The class method is actually an object method of the class object.

In NX, all classes are objects. Classes are in NX special kind of objects that have e.g. the ability to create instances and to provide methods for the instances. Classes manage their instances. The general method set for classes is defined on the meta-classes (more about this later).

The following example defines a public class method `bar` on class `C`. The class method is specified by using the modifier `object` in front of `method` in the method definition command.

Listing 30: Class Methods

```
nx::Class create C {
  #
  # Define a class method "bar" and an instance
  # method "foo"
  #
  :public object method bar {} {return 2}
  :public method foo {} {return 1}

  #
  # Create an instance of the current class
  #
  :create c1
}

# Method "bar" is a class method of class "C"
# therefore applicable on the class object "C"
C bar

# Method "foo" is an instance method of "C"
# therefore applicable on instance "c1"
c1 foo

# When trying to invoke the class method on the
# instance, an error will be raised.
c1 bar
```

In some other object-oriented programming languages, class methods are called "static methods".

3.5. Ensemble Methods

NX provides *ensemble methods* as a means to structure the method name space and to group related

methods. Ensemble methods are similar in concept to Tcl's ensemble commands.

> An **ensemble method** is a form of a hierarchical method consisting of a container method and sub-methods. The first argument of the container method is interpreted as a selector (the sub-method). Every sub-method can be an container method as well.

Ensemble methods provide a means to group related commands together, and they are extensible in various ways. It is possible to add sub-methods at any time to existing ensembles. Furthermore, it is possible to extend ensemble methods via mixin classes.

The following example defines an ensemble method for `string`. An ensemble method is defined when the provide method name contains a space.

Listing 31: Ensemble Method

```
nx::Class create C {

    # Define an ensemble method "string" with sub-methods
    # "length", "tolower" and "info"

    :public method "string length"  {x} {....}
    :public method "string tolower" {x} {...}
    :public method "string info" {x} {...}
    #...
    :create c1
}

# Invoke the ensemble method
c1 string length "hello world"
```

3.6. Method Resolution

When a method is invoked, the applicable method is searched in the following order:

Per-object Mixins -> Per-class Mixins -> Object -> Intrinsic Class Hierarchy

In the case, no mixins are involved, first the object is searched for an object method with the given name, and then the class hierarchy of the object. The method can be defined multiple times on the search path, so some of these method definitions might be *shadowed* by the more specific definitions.

Listing 32: Method Resolution with Intrinsic Classes

```
nx::Class create C {
  :public method foo {} {
    return "C foo: [next]"
  }
}

nx::Class create D -superclass C {

  :public method foo {} {
    return "D foo: [next]"
  }

   :create d1 {
     :public object method foo {} {
       return "d1 foo: [next]"
     }
   }
}

# Invoke the method foo
```

```
d1 foo
# result: "d1 foo: D foo: C foo: "

# Query the precedence order from NX via introspection
d1 info precedence
# result: "::D ::C ::nx::Object"
```

Consider the example in Listing 32. When the method `foo` is invoked on object `d1`, the object method has the highest precedence and is therefore invoked. The object methods shadows the same-named methods in the class hierarchy, namely the method `foo` of class `D` and the method `foo` of class `C`. The shadowed methods can be still invoked, either via the primitive `next` or via method handles (we used already method handles in the section about method aliases). In the example above, `next` calls the shadowed method and add their results to the results of every method. So, the final result contains parts from `d1`, `D` and `C`. Note that the topmost `next` in method `foo` of class `C` shadows no method `foo` and simply returns empty (and not an error message).

The introspection method `info precedence` provides information about the order, in which classes processed during method resolution.

Listing 33: Method Resolution with Mixin Classes

```
nx::Class create M1 {
  :public method foo {} { return "M1 foo: [next]"}
}
nx::Class create M2 {
  :public method foo {} { return "M2 foo: [next]"}
}

#
# "d1" is created based on the definitions of the last example
#
# Add the methods from "M1" as per-object mixin to "d1"
d1 object mixins add M1

#
# Add the methods from "M2" as per-class mixin to class "C"
C mixins add M2

# Invoke the method foo
d1 foo
# result: "M1 foo: M2 foo: d1 foo: D foo: C foo: "

# Query the precedence order from NX via introspection
d1 info precedence
# result: "::M1 ::M2 ::D ::C ::nx::Object"
```

The example in Listing 33 is an extension of the previous example. We define here two additional classes `M1` and `M2` which are used as per-object and per-class mixins. Both classes define the method `foo`, these methods shadow the definitions of the intrinsic class hierarchy. Therefore an invocation of `foo` on object `d1` causes first an invocation of method in the per-object mixin.

Listing 34: Method Invocation Flags

```
#
# "d1" is created based on the definitions of the last two examples,
# the mixins "M1" and "M2" are registered.
#
# Define a public object method "bar", which calls the method
# "foo" which various invocation options:
#
d1 public object method bar {} {
   puts [:foo]
   puts [: -local foo]
```

```
    puts [: -intrinsic foo]
    puts [: -system foo]
}

# Invoke the method "bar"
d1 bar
```

In the first line of the body of method `bar`, the method `foo` is called as usual with an implicit receiver, which defaults to the current object (therefore, the call is equivalent to `d1 foo`). The next three calls show how to provide flags that influence the method resolution. The flags can be provided between the colon and the method name. These flags are used rather seldom but can be helpful in some situations.

The invocation flag `-local` means that the method has to be resolved from the same place, where the current method is defined. Since the current method is defined as a object method, `foo` is resolved as a object method. The effect is that the mixin definitions are ignored. The invocation flag `-local` was already introduced int the section about method protection, where it was used to call *private* methods.

The invocation flag `-intrinsic` means that the method has to be resolved from the intrinsic definitions, meaning simply without mixins. The effect is here the same as with the invocation flag `-local`.

The invocation flag `-system` means that the method has to be resolved from basic - typically predefined - classes of the object system. This can be useful, when script overloads system methods, but still want to call the shadowed methods from the base classes. In our case, we have no definitions of `foo` on the base clases, therefore an error message is returned.

The output of Listing 34 is:

```
M1 foo: M2 foo: d1 foo: D foo: C foo:
d1 foo: D foo: C foo:
d1 foo: D foo: C foo:
::d1: unable to dispatch method 'foo'
```

3.7. Parameters

NX provides a generalized mechanism for passing values to either methods (we refer to these as *method parameters*) or to objects (these are called *configure parameters*). Both kind of parameters might have different features, such as:

- Positional and non-positional parameters
- Required and non-required parameters
- Default values for parameters
- Value-checking for parameters
- Multiplicity of parameters

TODO: complete list above and provide a short summary of the section

Before we discuss method and configure parameters in more detail, we describe the parameter features in the subsequent sections based on method parameters.

3.7.1. Positional and Non-Positional Parameters

If the position of a parameter in the list of formal arguments (e.g. passed to a function) is significant for its meaning, this is a *positional* parameter. If the meaning of the parameter is independent of its position, this is a *non-positional* parameter. When we call a method with positional parameters, the meaning of the parameters (the association with the argument in the argument list of the method) is

determined by its position. When we call a method with non-positional parameters, their meaning is determined via a name passed with the argument during invocation.

Listing 35: Positional and Non-Positional Method Parameters

```
nx::Object create o1 {

  #
  # Method foo has positional parameters:
  #
  :public object method foo {x y} {
    puts "x=$x y=$y"
  }

  #
  # Method bar has non-positional parameters:
  #
  :public object method bar {-x -y} {
    puts "x=$x y=$y"
  }

  #
  # Method baz has non-positional and
  # positional parameters:
  #
  :public object method baz {-x -y a} {
    puts "x? [info exists x] y? [info exists y] a=$a"
  }
}

# invoke foo (positional parameters)
o1 foo 1 2

# invoke bar (non-positional parameters)
o1 bar -y 3 -x 1
o1 bar -x 1 -y 3

# invoke baz (positional and non-positional parameters)
o1 baz -x 1 100
o1 baz 200
o1 baz -- -y
```

Consider the example in Listing 35. The method `foo` has the argument list `x y`. This means that the first argument is passed in an invocation like `o1 foo 1 2` to `x` (here, the value `1`), and the second argument is passed to `y` (here the value `2`). Method `bar` has in contrary just with non-positional arguments. Here we pass the names of the parameter together with the values. In the invocation `o1 bar -y 3 -x 1` the names of the parameters are prefixed with a dash ("-"). No matter whether in which order we write the non-positional parameters in the invocation (see line 30 and 31 in Listing 35) in both cases the variables `x` and `y` in the body of the method `bar` get the same values assigned (`x` becomes `1`, `y` becomes `3`).

It is certainly possible to combine positional and non-positional arguments. Method `baz` provides two non-positional parameter (`-y` and `-y`) and one positional parameter (namely `a`). The invocation in line 34 passes the value of `1` to `x` and the value of `100` to `a`. There is no value passed to `y`, therefore value of `y` will be undefined in the body of `baz`, `info exists y` checks for the existence of the variable `y` and returns `0`.

The invocation in line 35 passes only a value to the positional parameter. A more tricky case is in line 36, where we want to pass `-y` as a value to the positional parameter `a`. The case is more tricky since syntactically the argument parser might consider `-y` as the name of one of the non-positional parameter. Therefore we use `--` (double dash) to indicate the end of the block of the non-positional parameters and therefore the value of `-y` is passed to `a`.

3.7.2. Optional and Required Parameters

Per default positional parameters are required, and non-positional parameters are optional (they can be left out). By using parameter options, we can as well define positional parameters, which are optional, and non-positional parameters, which are required.

Listing 36: Optional and Required Method Parameters

```
nx::Object create o2 {

  #
  # Method foo has one required and one optional
  # positional parameter:
  #
  :public object method foo {x:required y:optional} {
    puts "x=$x y? [info exists y]"
  }

  #
  # Method bar has one required and one optional
  # non-positional parameter:
  #
  :public object method bar {-x:required -y:optional} {
    puts "x=$x y? [info exists y]"
  }
}

# invoke foo (one optional positional parameter is missing)
o2 foo 1
```

The example in Listing 36 defined method `foo` with one required and one optional positional parameter. For this purpose we use the parameter options `required` and `optional`. The parameter options are separated from the parameter name by a colon. If there are multiple parameter options, these are separated by commas (we show this in later examples).

The parameter definition `x:required` for method `foo` is equivalent to `x` without any parameter options (see e.g. previous example), since positional parameters are per default required. The invocation in line 21 of Listing 36 will lead to an undefined variable `y` in method `foo`, because no value us passed to the optional parameter. Note that only trailing positional parameters might be optional. If we would call method `foo` of Listing 35 with only one argument, the system would raise an exception.

Similarly, we define method `bar` in Listing 36 with one required and one optional non-positional parameter. The parameter definition `-y:optional` is equivalent to `-y`, since non-positional parameter are per default optional. However, the non-positional parameter `-x:required` is required. If we invoke `bar` without it, the system will raise an exception.

3.7.3. Default Values for Parameters

Optional parameters might have a default value. This default value is used, when no argument is provided for the corresponding parameter. Default values can be specified for positional and non-positional parameters.

Listing 37: Method Parameters with Default Values

```
nx::Object create o3 {

  #
  # Positional parameter with default value:
  #
  :public object method foo {{x 1} {y 2}} {
    puts "x=$x y=$y"
  }
```

```
  #
  # Non-positional parameter with default value:
  #
  :public object method bar {{-x 10} {-y 20}} {
    puts "x=$x y=$y"
  }
}

# use default values
o3 foo
o3 bar
```

In order to define a default value for a parameter, the parameter specification must be of the form of a 2 element list, where the second argument is the default value. See for an example in Listing 37.

3.7.4. Value Constraints

NX provides value constraints for all kind of parameters. By specifying value constraints a developer can restrict the permissible values for a parameter and document the expected values in the source code. Value checking in NX is conditional, it can be turned on or off in general or on a per-usage level (more about this later). The same mechanisms can be used not only for input value checking, but as well for return value checking (we will address this point as well later).

Built-in Value Constraints

NX comes with a set of built-in value constraints, which can be extended on the scripting level. The built-in checkers are either the native checkers provided directly by the Next Scripting Framework (the most efficient checkers) or the value checkers provided by Tcl through `string is` ... The built-in checkers have as well the advantage that they can be used also at any time during bootstrap of an object system, at a time, when e.g. no objects or methods are defined. The same checkers are used as well for all C-implemented primitives of NX and the Next Scripting Framework.

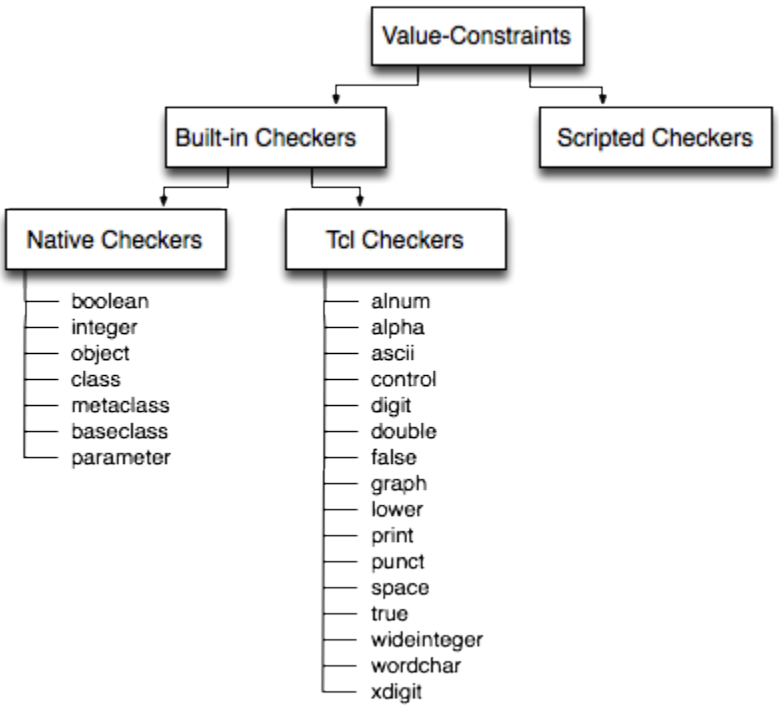

Figure 38. General Applicable Value Checkers in NX

Figure 38 shows the built-in general applicable value checkers available in NX, which can be used for all method and configure parameters. In the next step, we show how to use these value-checkers for checking permissible values for method parameters. Then we will show, how to provide more detailed value constraints.

Listing 39: Method Parameters with Value Constraints

```
nx::Object create o4 {

  #
  # Positional parameter with value constraints:
  #
  :public object method foo {x:integer o:object,optional} {
    puts "x=$x o? [info exists o]"
  }

  #
  # Non-positional parameter with value constraints:
  #
  :public object method bar {{-x:integer 10} {-verbose:boolean false}} {
    puts "x=$x verbose=$verbose"
  }
}

# The following invocation raises an exception, since the
# value "a" for parameter "x" is not an integer
o4 foo a
```

Value constraints are specified as parameter options in the parameter specifications. The parameter specification `x:integer` defines `x` as a required positional parameter which value is constraint to an integer. The parameter specification `o:object,optional` shows how to combine multiple parameter options. The parameter `o` is an optional positional parameter, its value must be an object (see Listing 39). Value constraints are specified exactly the same way for non-positional parameters (see method `bar` in Listing 39).

Listing 40: Parameterized Value Constraints

```
#
# Create classes for Person and Project
#
nx::Class create Person
nx::Class create Project

nx::Object create o5 {
  #
  # Parameterized value constraints
  #
  :public object method work {
     -person:object,type=Person
     -project:object,type=Project
   } {
    # ...
  }
}

#
# Create a Person and a Project instance
#
Person create gustaf
Project create nx

#
# Use method with value constraints
#
o5 work -person gustaf -project nx
```

The native checkers `object`, `class`, `metaclass` and `baseclass` can be further specialized with the parameter option `type` to restrict the permissible values to instances of certain classes. We can use for example the native value constraint `object` either for testing whether an argument is some object (without further constraints, as in Listing 37, method `foo`), or we can constrain the value further to some type (direct or indirect instance of a class). This is shown by method `work` in Listing 40 which requires the parameter `-person` to be an instance of class `Person` and the parameter `-project` to be an instance of class `Project`.

Scripted Value Constraints

The set of predefined value checkers can be extended by application programs via defining methods following certain conventions. The user defined value checkers are defined as methods of the class `nx::Slot` or of one of its subclasses or instances. We will address such cases in the next sections. In the following example we define two new value checkers on class `nx::Slot`. The first value checker is called `groupsize`, the second one is called `choice`.

Listing 41: Scripted Value Checker for Method Parameters

```
#
# Value checker named "groupsize"
#
::nx::Slot method type=groupsize {name value} {
  if {$value < 1 || $value > 6} {
    error "Value '$value' of parameter $name is not between 1 and 6"
  }
}

#
# Value checker named "choice" with extra argument
#
::nx::Slot method type=choice {name value arg} {
  if {$value ni [split $arg |]} {
    error "Value '$value' of parameter $name not in permissible values $arg"
  }
}

#
# Create an application class D
# using the new value checkers
#
nx::Class create D {
  :public method foo {a:groupsize} {
    # ...
  }
  :public method bar {a:choice,arg=red|yellow|green b:choice,arg=good|bad} {
    # ...
  }
}

D create d1

# testing "groupsize";
# the second call (with value 10) will raise an exception:
d1 foo 2
d1 foo 10

# testing "choice"
# the second call (with value pink for parameter a)
# will raise an exception:
d1 bar green good
d1 bar pink bad
```

In order to define a checker `groupsize` a method of the name `type=groupsize` is defined. This method receives two arguments, `name` and `value`. The first argument is the name of the parameter

(mostly used for the error message) and the second parameter is provided value. The value checker simply tests whether the provided value is between 1 and 3 and raises an exception if this is not the case (invocation in line 36 in Listing 41).

The checker `groupsize` has the permissible values defined in its method's body. It is as well possible to define more generic checkers that can be parameterized. For this parameterization, one can pass an argument to the checker method (last argument). The checker `choice` can be used for restricting the values to a set of predefined constants. This set is defined in the parameter specification. The parameter `a` of method `bar` in Listing 41 is restricted to the values `red`, `yellow` or `green`, and the parameter `b` is restricted to `good` or `bad`. Note that the syntax of the permissible values is solely defined by the definition of the value checker in lines 13 to 17. The invocation in line 39 will be ok, the invocation in line 40 will raise an exception, since `pink` is not allowed.

If the same checks are used in many places in the program, defining names for the value checker will be the better choice since it improves maintainability. For seldom used kind of checks, the parameterized value checkers might be more convenient.

3.7.5. Multiplicity

Multiplicity is used to define whether a parameter should receive single or multiple values.

A multiplicity specification has a lower and an upper bound. A lower bound of `0` means that the value might be empty. A lower bound of `1` means that the parameter needs at least one value. The upper bound might be `1` or `n` (or synonymously `*`). While the upper bound of `1` states that at most one value has to be passed, the upper bound of `n` says that multiple values are permitted. Other kinds of multiplicity are currently not allowed.

The multiplicity is written as parameter option in the parameter specification in the form *lower-bound..upper-bound*. If no multiplicity is defined the default multiplicity is `1..1`, which means: provide exactly one (atomic) value (this was the case in the previous examples).

Listing 42: Method Parameters with Explicit Multiplicity

```
nx::Object create o6 {

  #
  # Positional parameter with an possibly empty
  # single value
  #
  :public object method foo {x:integer,0..1} {
    puts "x=$x"
  }

  #
  # Positional parameter with an possibly empty
  # list of values value
  #
  :public object method bar {x:integer,0..n} {
    puts "x=$x"
  }

  #
  # Positional parameter with a non-empty
  # list of values
  #
  :public object method baz {x:integer,1..n} {
    puts "x=$x"
  }
}
```

Listing 42 contains three examples for positional parameters with different multiplicities. Multiplicity is often combined with value constraints. A parameter specification of the form `x:integer,0..n` means that the parameter `x` receives a list of integers, which might be empty. Note that the value constraints are applied to every single element of the list.

The parameter specification `x:integer,0..1` means that `x` might be an integer or it might be empty. This is one style of specifying that no explicit value is passed for a certain parameter. Another style is to use required or optional parameters. NX does not enforce any particular style for handling unspecified values.

All the examples in Listing 42 are for single positional parameters. Certainly, multiplicity is fully orthogonal with the other parameter features and can be used as well for multiple parameters, non-positional parameter, default values, etc.

3.7.6. Defaults substitution

Optional object and method parameters can set a default value. Recall that default values can be specified for positional and non-positional parameters, alike. This default value is used to define a corresponding method-local and object variable, respectively, and to set it to the default value. By default, the default value is taken literally (without any substitutions). Default values can also be preprocessed into a final value using Tcl substitution as provided by the Tcl `[subst]` command. To control the kind of substitutions to be performed, the parameter option `substdefault` can be provided.

Listing 43: Default-value substitution using `substdefault`

```
nx::Class create ::D
nx::Class create ::C {
  #
  # By default all substitutions (command, variable, control
  # characters) are active, when "substdefault" is used:
  #
  :property {d:object,type=::D,substdefault {[::D new]}}

  #
  # The actual property values are computed and
  # set at instantiation time.
  #
  :create ::c
}

::c cget -d
```

Listing 43 uses `substdefault` to provide a default value for the property `d`. In this example, the default value is a fresh instance of class `::D`. When the parameter option `substdefault` is used default, all substitution kinds of Tcl are active: command, variable, and backslash substitution. `substdefault` can be parametrized to include or to exclude any combination of substitution kinds by providing a bitmask:

- `substdefault=0b111`: all substitutions active (default)
- `substdefault=0b100`: substitute backslashes only (like `subst -novariables -nocommands`)
- `substdefault=0b010`: substitute variables only (like `subst -nobackslashes -nocommands`)
- `substdefault=0b001`: substitute commands only (like `subst -nobackslashes -novariables`)
- `substdefault=0b000`: substitute nothing (like `subst -nobackslashes -nocommands -novariables`, noop)

4. Advanced Language Features

...

4.1. Objects, Classes and Meta-Classes

...

4.2. Resolution Order and Next-Path

...

4.3. Details on Method and Configure Parameters

The parameter specifications are used in NX for the following purposes. They are used for

- the specification of input arguments of methods and commands, for
- the specification of return values of methods and commands, and for
- the specification for the initialization of objects.

We refer to the first two as method parameters and the last one as configure parameters. The examples in the previous sections all parameter specification were specifications of method parameters.

> **Method parameters** specify properties about permissible values passed to methods.

The method parameter specify how methods are invoked, how the actual arguments are passed to local variables of the invoked method and what kind of checks should be performed on these.

> **Configure parameters** are parameters that specify, how objects can be parameterized upon creation.

Syntactically, configure parameters and method parameters are the same, although there are certain differences (e.g. some parameter options are only applicable for objects parameters, the list of object parameters is computed dynamically from the class structures, object parameters are often used in combination with special setter methods, etc.). Consider the following example, where we define the two application classes `Person` and `Student` with a few properties.

Listing 44: Configure Parameters

```
#
# Define a class Person with properties "name"
# and "birthday"
#
nx::Class create Person {
  :property name:required
  :property birthday
}
```

```
#
# Define a class Student as specialization of Person
# with and additional property
#
nx::Class create Student -superclass Person {
  :property matnr:required
  :property {oncampus:boolean true}
}

#
# Create instances using configure parameters
# for the initialization
#
Person create p1 -name Bob
Student create s1 -name Susan -matnr 4711

# Access property value via "cget" method
puts "The name of s1 is [s1 cget -name]"
```

The class `Person` has two properties `name` and `birthday`, where the property `name` is required, the property `birthday` is not. The class `Student` is a subclass of `Person` with the additional required property `matnr` and an optional property `oncampus` with the default value `true` (see Listing 44). The class diagram below visualizes these definitions.

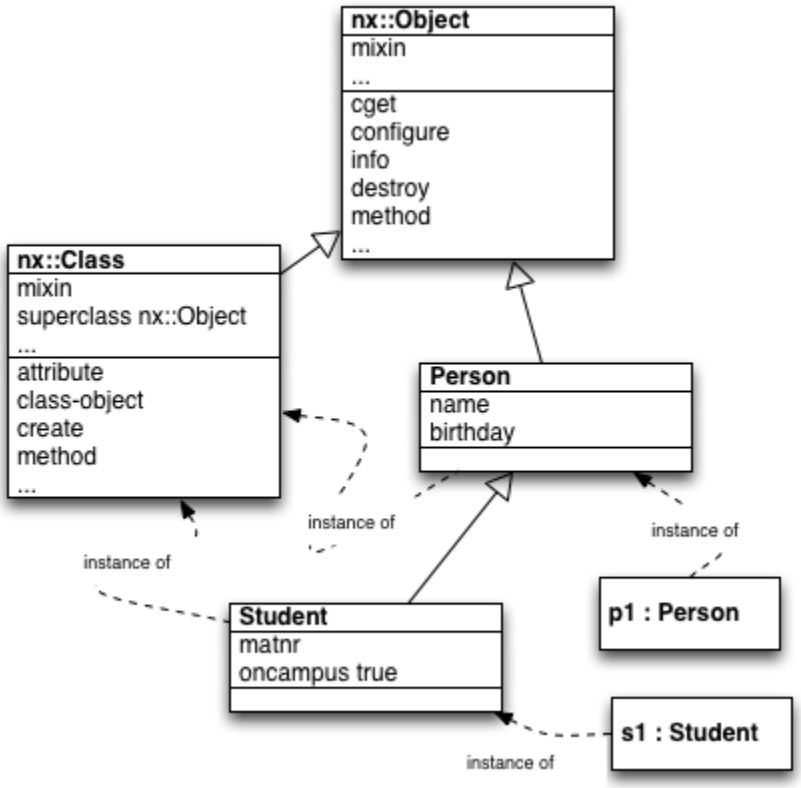

Figure 45. System and Application Classes

In NX, these definitions imply that instances of the class of `Person` have the properties `name` and `birthday` as *non-positional object parameters*. Furthermore it implies that instances of `Student` will have the configure parameters of `Person` augmented with the object parameters from `Student` (namely `matnr` and `oncampus`). Based on these configure parameters, we can create a `Person` named `Bob` and a `Student` named `Susan` with the matriculation number `4711` (see line 23 and 24 in <<xmp-object-parameters, instance variables `name`, `matnr` and `oncampus` (the latter is initialized with the

default value).

4.3.1. Configure Parameters available for all NX Objects

The configure parameters are not limited to the application defined properties, also NX provides some predefined definitions. Since `Person` is a subclass of `nx::Object` also the configure parameters of `nx::Object` are inherited. In the introductory stack example, we used `-mixins` applied to an object to denote per-object mixins (see Listing 8). Since `mixins` is defined as a parameter on `nx::Object` it can be used as an object parameter `-mixins` for all objects in NX. To put it in other words, every object can be configured to have per-object mixins. If we would remove this definition, this feature would be removed as well.

As shown in the introductory examples, every object can be configured via a scripted initialization block (the optional scripted block specified at object creation as last argument; see Listing 5 or Listing 12). The scripted block and its meaning are as well defined by the means of configure parameters. However, this configure parameter is positional (last argument) and optional (it can be omitted). The following listing shows the configure parameters of `Person p1` and `Student s1`.

Listing 46: Computed Actual Configure Parameter

```
Configure parameters for Person p1:
   Command:
      p1 info lookup syntax configure
   Result:
      -name /value/ ?-birthday /value/? ?-object-mixins /mixinreg .../?
      ?-class /class/? ?-object-filters /filterreg .../? ?/__initblock/?

Configure parameter for Student s1:
   Command:
      s1 info lookup syntax configure
   Result:
      ?-oncampus /boolean/? -matnr /value/ -name /value/
      ?-birthday /value/? ?-object-mixins /mixinreg .../? ?-class /class/?
      ?-object-filters /filterreg .../? ?/__initblock/?
```

The given parameter show, how (a) objects can be configured at runtime or (b) how new instances can be configured at creation time via the `new` or `create` methods. Introspection can be used to obtain the configuration parameters from an object via `p1 info lookup parameters configure` (returning the configure parameters currently applicable for `configure` or `cget`) or from a class `Person info lookup parameters create` on a class (returning the configure parameters applicable when an object of this class is created)

The listed configure parameter types `mixinreg` and `filterreg` are for converting definitions of filters and mixins. The last value `__initblock` says that the content of this variable will be executed in the context of the object being created (before the constructor `init` is called). More about the configure parameter types later.

4.3.2. Configure Parameters available for all NX Classes

Since classes are certain kind of objects, classes are parameterized in the same way as objects. A typical parameter for a class definition is the relation of the class to its superclass.In our example, we have specified, that `Student` has `Person` as superclass via the non-positional configure parameter `-superclass`. If no superclass is specified for a class, the default superclass is `nx::Object`. Therefore `nx::Object` is the default value for the parameter `superclass`.

Another frequently used parameter for classes is `-mixins` to denote per-class mixins (see e.g. the introductory Stack example in Listing 10), which is defined in the same way.

Since `Student` is an instance of the meta-class `nx::Class` it inherits the configure parameters from `nx::Class` (see class diagram Figure 45). Therefore, one can use e.g. `-superclass` in the definition

of classes.

Since `nx::Class` is a subclass of `nx::Object`, the meta-class `nx::Class` inherits the parameter definitions from the most general class `nx::Object`. Therefore, every class might as well be configured with a scripted initialization block the same way as objects can be configured. We used actually this scripted initialization block in most examples for defining the methods of the class. The following listing shows (simplified) the parameters applicable for `Class Student`.

Listing 47: Parameters for Classes

```
Configure parameter for class nx::Class
   Command:
      nx::Class info lookup syntax configure
   Result:
      ?-superclass /class .../? ?-mixins /mixinreg .../?
      ?-filters /filterreg .../? ?-object-mixins /mixinreg .../?
      ?-class /class/? ?-object-filters /filterreg .../? ?/__initblock/?
```

4.3.3. User defined Parameter Types

More detailed definition of the configure parameter types comes here.

4.3.4. Slot Classes and Slot Objects

In one of the previous sections, we defined scripted (application defined) checker methods on a class named `nx::Slot`. In general NX offers the possibility to define value checkers not only for all usages of parameters but as well differently for method parameters or configure parameters

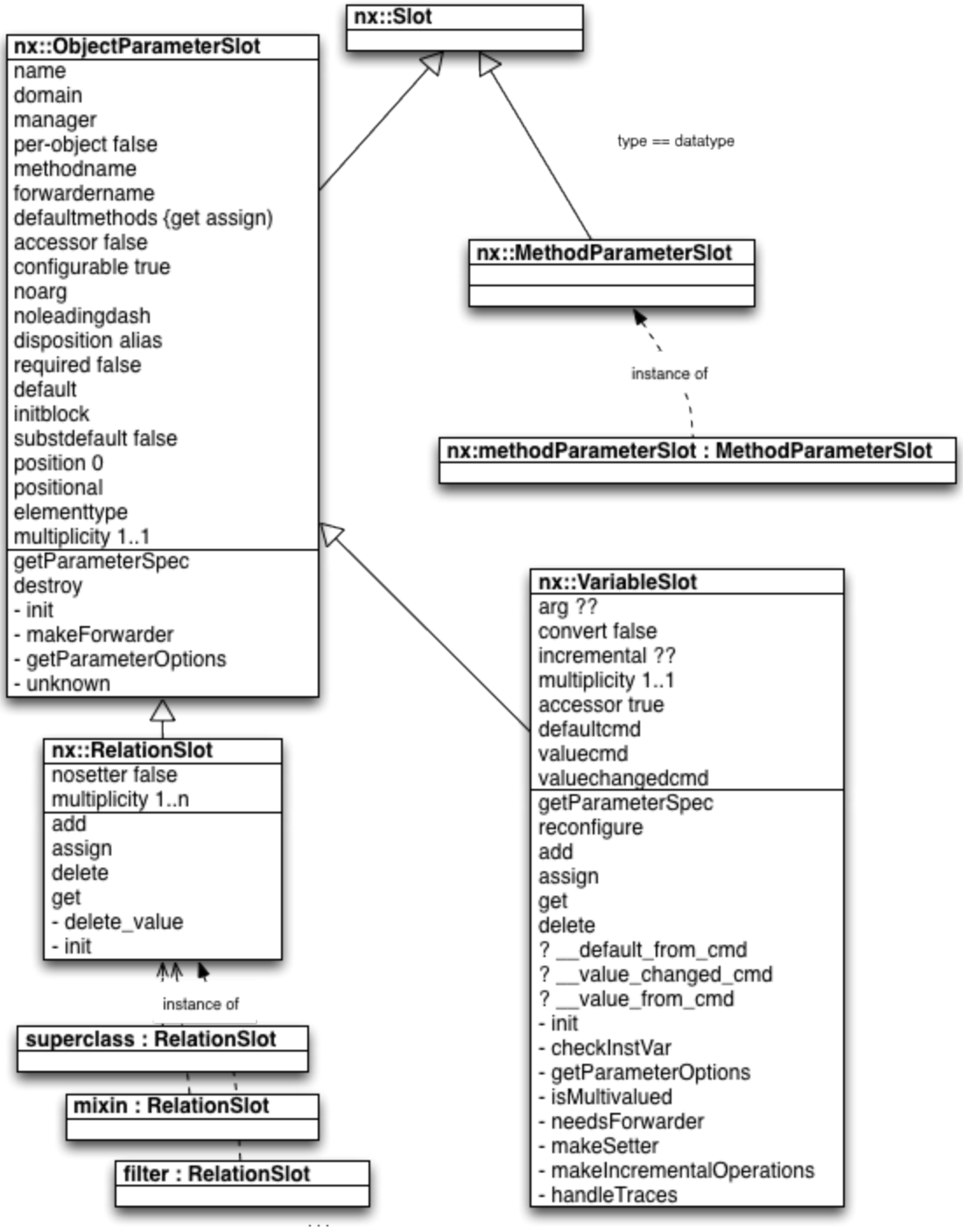

Figure 48. Slot Classes and Objects

4.3.5. Attribute Slots

Still Missing

- return value checking
- switch
- initcmd ...
- subst rules

- converter
- incremental slots

5. Miscellaneous

...

5.1. Profiling

...

5.2. Unknown Handlers

NX provides two kinds of unknown handlers:

- Unknown handlers for methods
- Unknown handlers for objects and classes

5.2.1. Unknown Handlers for Methods

Object and classes might be equipped with a method `unknown` which is called in cases, where an unknown method is called. The method unknown receives as first argument the called method followed by the provided arguments

Listing 49: Unknown Method Handler

```
::nx::Object create o {
  :object method unknown {called_method args} {
    puts "Unknown method '$called_method' called"
  }
}

# Invoke an unknown method for object o:
o foo 1 2 3

# Output will be: "Unknown method 'foo' called"
```

Without any provision of an unknown method handler, an error will be raised, when an unknown method is called.

5.2.2. Unknown Handlers for Objects and Classes

The next scripting framework provides in addition to unknown method handlers also a means to dynamically create objects and classes, when these are referenced. This happens e.g. when superclasses, mixins, or parent objects are referenced. This mechanism can be used to implement e.g. lazy loading of these classes. Nsf allows one to register multiple unknown handlers, each identified by a key (a unique name, different from the keys of other unknown handlers).

Listing 50: Unknown Class Handler

```
::nx::Class public object method __unknown {name} {
  # A very simple unknown handler, showing just how
  # the mechanism works.
```

```
  puts "***** __unknown called with <$name>"
  ::nx::Class create $name
}

# Register an unknown handler as a method of ::nx::Class
::nsf::object::unknown::add nx {::nx::Class __unknown}

::nx::Object create o {
  # The class M is unknown at this point

  :object mixins add M
  # The line above has triggered the unknown class handler,
  # class M is now defined

  puts [:info object mixins]
  # The output will be:
  #     ***** __unknown called with <::M>
  #     ::M
}
```

The Next Scripting Framework allows one to add, query, delete and list unknown handlers.

Listing 51: Unknown Handler registration

```
# Interface for unknown handlers:
# nsf::object::unknown::add /key/ /handler/
# nsf::object::unknown::get /key/
# nsf::object::unknown::delete /key/
# nsf::object::unknown::keys
```

References

- U. Zdun, M. Strembeck, G. Neumann: Object-Based and Class-Based Composition of Transitive Mixins, Information and Software Technology, 49(8) 2007 .
- G. Neumann and U. Zdun: Filters as a language support for design patterns in object-oriented scripting languages. In Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems, San Diego, May 1999.
- G. Neumann and U. Zdun: Implementing object-specific design patterns using per-object mixins. In Proc. of NOSA`99, Second Nordic Workshop on Software Architecture, Ronneby, Sweden, August 1999.
- G. Neumann and U. Zdun: Enhancing object-based system composition through per-object mixins. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Takamatsu, Japan, December 1999.
- G. Neumann and U. Zdun: XOTCL, an object-oriented scripting language. In Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, February 2000.
- G. Neumann and U. Zdun: Towards the Usage of Dynamic Object Aggregations as a Form of Composition In: Proceedings of Symposium of Applied Computing (SAC'00), Como, Italy, Mar 19-21, 2000.
- G. Neumann, S. Sobernig: XOTcl 2.0 - A Ten-Year Retrospective and Outlook, in: Proceedings of the Sixteenth Annual Tcl/Tk Conference, Portland, Oregon, October, 2009.
- J. K. Ousterhout: Tcl: An embeddable command language. In Proc. of the 1990 Winter USENIX Conference, January 1990.
- J. K. Ousterhout: Scripting: Higher Level Programming for the 21st Century, IEEE Computer 31(3), March 1998.
- D. Wetherall and C. J. Lindblad: Extending Tcl for Dynamic Object-Oriented Programming. Proc. of the Tcl/Tk Workshop '95, July 1995.

Version 2.3.0
Last updated 2019-05-07 11:27:03 CEST