# New scope mechanisms for the Smalltalk-80 language

L. Peter Deutsch
Xerox, ParcPlace Systems

This is a working document proposing mechanisms for more flexible control of name scopes in the Smalltalk-80 language, including private and nested name spaces for classes.

Comments are solicited. For changes in successive versions, consult the version history.

Version history:

[2] 7 May 1987: split off this document (scopes) from pkg87; incorporated comments from Richard Steiger, Mark Miller, and February implementors' workshop (no substantive changes, only clarifications).

[1] 25 February 1987: first version (pkg87), distributed to implementors at the Feb. 26-27 workshop.

---

## Introduction

Conventional language systems support independent development of application packages with encapsulation mechanisms (such as name hiding) that allow them to be combined into a single executable program. The Smalltalk-80 system currently does not provide facilities, which substantially hinders the ability of developers to produce widely-usable packages. This document proposes a more capable name scoping facility for the Smalltalk-80 system. It does not address the other problem of encapsulation, namely how to create truly protected objects using the name scoping facilities in concert with object-oriented design.

In theory, an object-oriented language such as Smalltalk provides excellent modularity, since messages are interpreted relative to the class. In practice, there are several barriers to meeting this promise:

- All class names are global, so any object can send a message to any class. This makes protection awkward, because typical applications have both public and private classes. It also raises a problem of name clashes between classes of different applications.
- The system facilities have been optimized for development environments, where it is very valuable to have all relevant symbolic information readily available (such as message and variable names), as opposed to delivery environments, where it may be more desirable (for reasons of both space and protection) to do without this information.

We propose to address these problems through a generalization of the present class variable / pool dictionary mechanism, which allows the easy creation of private scopes for static variables and constants (including classes).

Implementing the proposal would involve very minor changes to the Smalltalk compiler, and minor changes to the Debugger and Inspector.

The present proposal is designed primarily for helping to organize large-scale systems composed of many weakly interacting application areas, i.e. it is aimed primarily at the problem of name conflicts.

## Scopes
------

The present Smalltalk-80 language has four kinds of instantiable variables (variables which can exist in multiple instances):

- Block variables (arguments and temporaries);
- Method variables (arguments and temporaries);
- Instance variables;
- Class instance variables.

The present system has three kinds of static variables (variables which only exist in a single instance):

- Class variables;
- Pool variables;
- Global variables (in Smalltalk or Undeclared).

We believe that it is both feasible and desirable to collapse these three kinds of static variables into a single notion of a static variable dictionary: the proposals below indicate how to do this at an implementation level, but do not specify what effect it might have on the language.

In the long term, we believe that it is both feasible and highly desirable to reduce the number of conceptually different kinds of variable, ideally to a single concept of an instantiable variable and instantiable scope. However, at this time we have no proposal for doing this. In particular, we are not prepared to propose eliminating the instantiable / static distinction, even though static variables are conceptually at odds with the notion of instantiable scopes.

Independent of this issue, we observe that in order for the Debugger to evaluate expressions relative to a particular context, it must be able to invoke the compiler and pass it extra information describing how to interpret and resolve names referring to local variables of the context. We also note that the Tektronix Smalltalk system provides a facility for defining static variables local to individual workspaces: this also implies the ability to compile (or at least evaluate) expressions in a specified naming context. We propose here to extend the compiler interface in a way that allows programmers to construct arbitrary scope structures involving static variables. This solves the problem at the implementation level, but not the language level: we do not address the latter problem here.

The key concept in our approach to more flexible name resolutions is to identify an abstract class NameScope and its protocol. In the present version of this proposal, we only deal with static scopes, i.e. those for which supplying only the name is sufficient to obtain a value, without needing any context information. We hope to extend this to include instantiable scopes in the future.

Conceptually, a NameScope is an ordered collection of dictionaries. Dictionaries earlier in the ordering take precedence over those appearing later, if the same name appears in more than one. (Allowing the same name to appear more than once is important, since otherwise one might have to change names in a local scope as a result of something happening in an outer scope in which one has no interest.) However, a NameScope behaves (through its public protocol) like a single dictionary, i.e. as though it were a dictionary in which the keys are all the names recognized within the scope. For example, an evaluator can use the at: or at:ifAbsent: message to obtain the value of a variable; a compiler can use the associationAt: message to obtain a reference to the binding of a name.

The only major public protocol distinction between NameScopes and Dictionaries is how they are created. In particular, NameScopes are created using the following messages:

<StaticNameScope> := <NameScope class> empty

Returns a NameScope that recognizes no variables.

<StaticNameScope> := <StaticNameScope> precededBy: <Dictionary>

Returns a new NameScope which looks up names in the Dictionary before looking them up in the receiver. In other words, if we think of a NameScope as a sequence of Dictionaries, in which Dictionaries appearing earlier in the list take precedence over those appearing later, this message produces a new NameScope with the Dictionary added at the head of the list. For example, the standard Smalltalk-80 global name scope is produced by ((NameScope empty precededBy: Undeclared) precededBy: Smalltalk). The Dictionaries are incorporated into the NameScope by reference in the following sense:

- If a name is added or removed in a Dictionary, subsequent compilations using any NameScope that includes that Dictionary will take account of the change.
- If a name is added or removed in a Dictionary, already-compiled methods that reference variables in that dictionary MAY OR MAY NOT take account of the change. In other words, we do NOT require that static variables be looked up in a chain of Dictionaries at the moment of use, the way that messages are looked up in MethodDictionaries at the moment of sending.

<InstantiableNameScope> := <Context> nameScope

Returns a NameScope appropriate for evaluating expressions in the given Context.

<ExecutableMethod> := <Behavior> compile: <String> in: <NameScope> ifError: <BlockClosure>

Compiles the String as a method for the given Behavior, using the NameScope as the outer context for names not defined in the Behavior or its superclasses. Note that the NameScope may have been obtained from a Context: this makes it possible, for example, to compile a new block relative to an existing context. This is also the first step in evaluating expressions relative to a context. We do not specify what happens if a name is not defined in the NameScope: we encourage implementors to provide a "soft" failure alternative such as the Undeclared dictionary provided in the present Smalltalk-80 system.

Using this mechanism, an application can construct classes whose names are entirely local to that application. Obviously one wants to have a linguistic mechanism for this, not just a set of procedural or even interactive tools. This is partly for user convenience, but also because there is a protection issue here: the messages for constructing and accessing scopes, and compiling, must be protected from unauthorized use. This question of so-called "meta-level" access is beyond the scope of this document.