Part IV: The Smalltalk Language

Questions about	Page
Literals	47
Variables and Names	53
Global Variables	59
Classes	65
What Variables Hold	68
Blocks	71
Methods	78
Inheritance, Self and Super	80
Other Questions	84

Literals

Literals are objects written literally, that is, with actual characters on the page or screen. Since the value of a literal is always the same, literals are sometimes called constants or literal constants.

Smalltalk has literal constants for integers, floating point numbers, a limited set of fractions, characters, booleans, strings, symbols, and arrays with literal contents.

Frequently Asked Question

4.1 • What are the forms of integer literals?

Integers have a rich set of literal constant forms which allow the description of values with bases ranging from 2 to 36 and in arbitrary lengths.

The simplest integer literal consists of one or more decimal digits with an optional leading minus sign.

```
1   123   -3
```

There is no practical limit to the length of an integer constant.†

```
124876435876348763498763498764359876459873659876134876234876132
```

A number base can be specified by a prefix with a decimal number in the range 2 to 36 followed by a lower case 'r', and followed by one or more of the digits allowed for that base. The values in each line below are equal:

```
2r11111111   4r3333   8r377   10r256   16rFF   32r7V
36rSMALLTALK   80738163270632
```

The characters allowed for base *n* are the *n* leading characters in this string:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Integers may also have an exponent, a lowercase 'e' followed by one or more decimal digits. The values in each line are equal:

```
1     1e0
300   2r11e2   3e2   4r3e2
```

See *4.3 'What are the forms of fraction literals?'* for information on negative exponents.

Note the following cases which do not produce quite the results that might be initially expected:

`16r2E1`	Produces the value `737`; the 'E' is uppercase and isn't an exponent.
`16RFF`	Tries to send the `RFF` message to `16` since the 'R' is uppercase, or get a message about digit too big, depending on the vendor.
`16r2abff`	Tries to send the `abff` message to `2`; 'abff' is lowercase.

Frequently Asked Question

4.2 • What are the forms of floating point literals?

Floating point numbers have a rich set of literal constant forms which allow the description of values with bases ranging from 2 to 36 and, in some implementations, in several precisions.

The simplest floating literal consists of one or more decimal digits with an optional leading minus sign and an *imbedded* decimal point.

```
1.0   12.3   -3.0
```

---

†. The actual limits depend on the implementation. A typical implementation might keep a 16-bit or a 32-bit count of the number of 32-bit words which make up the long integer. Execution time and memory size are the real limiting factors.

A leading or trailing decimal point is taken as a statement separator. These are *not* floating point constants:

```
123.   .123
```

There is no practical limit to the length of an floating point constant†, but only as much of the constant as can be represented in the implementation will be retained at run time. It is sometimes useful to code constants at a greater precision to allow for portability to platforms with a greater precision at a later date.

```
3.14159265358979323846264338327950288419716939937510582097494459
```

A number base can be specified by a prefix with a decimal number in the range 2 to 36 followed by a lower case 'r', and followed by one or more of the digits allowed for that base. The values in each line are equal:

```
2r11111111.0  4r3333.0  8r377.0  10r256.0  16rFF.0   32r7V.0
36rSMALL.TALK   48069417.81373362
```

The characters allowed for base *n* are the *n* leading characters in this string:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Floating point values may also have an exponent, which is a lowercase 'e' (and 'd' in ParcPlace implementations) followed by one or more decimal digits. The values in each line are equal:

```
1.0     1.0e0
300.0   2r11.0e2   3.0e2   4r3.0e2
```

In ParcPlace implementations, the `e` exponent indicates a short precision floating point number and the `d` indicates a double-precision floating-point number.

In other implementations, the precision is platform and implementation dependent.

See "Floating Point" on page 121 for more information on floating point numbers.

†. But, since only the first 70 or so digits are meaningful even if a quad-length value is supported, an implementation might wellnote have tested anything much longer.

Frequently Asked Question

4.3 • What are the forms of fraction literals?

The only fraction literals are almost accidental and come from allowing signed exponents on integers.

`123e-4` Equivalent to the expression: `(123/1000)`

Frequently Asked Question

4.4 • What is the form of a character literal?

Character literals consist of a dollar sign followed by a character. A dollar sign followed by a blank is the blank character, and followed by a dollar sign is a dollar sign.

`$a` `$A` `$z` `$9` `$.` `$(` `$)` `$ ` (a blank) `$$`

The character following the dollar sign can be any character in the implementation character set, including linefeed or return, but only the ASCII characters in the range 32 to 127 are guaranteed portable between implementations and platforms.

Frequently Asked Question

4.5 • What is the form of a string literal?

A string literal is a pair of single quote marks, possibly enclosing other characters.

```
''   'a'   'A String of Characters'
```

A single quote mark is represented by two successive single quote marks.

```
'It isn''t too hard to do.'
```

Note that a string with one character is not a character. These are not equal:

`'a'` `$a` **Example 1, Display**

Frequently Asked Question

4.6 • What are the boolean literals?

The boolean literals are `true` and `false`.

The value `true` is not the same as the class `True`; it is an instance of class `True`. The value `false` is not the same as the class `False`; it is an instance of class `False`. Thus, this code will fail with a message about the receiver not being a Boolean:

```
| bool |
bool := False.
bool ifFalse: [ Transcript cr; show: 'False']
```

Example 2, Display

The correct code is:

```
| bool |
bool := false.
bool ifFalse: [ Transcript cr; show: 'false']
```

Example 3, Display

While `true` is an instance of `True`, it is a special one and other instances of `True` cannot be substituted. The same also holds true for `false`.

```
" A new instance of True "
True basicNew ifTrue: [ 1 ]  ifFalse: [0]
" Gets message about not being a Boolean "
```

Example 4, Display

```
" A new instance of True "
True basicNew = true
" Answers false "
```

Example 5, Display

Frequently Asked Question

4.7 • What are the forms of symbol literals?

Symbols are similar to strings but have additional properties.

See (some question about what symbols really are...)

Symbol literals are made up of a pound or number sign ('`#`') followed by a message selector.

```
#==   #abc   #negated   #to:by:   #in:the:course:of:
```

Since valid binary selectors are made up of one or two special characters, the effect of writing three or more characters is implementation defined. For example, IBM Smalltalk takes `#===` as the two successive symbols `#==` and `#=` while Digitalk Smalltalk/V-Mac considers #=== as error.

IBM Smalltalk

IBM Smalltalk allows an alternate form of symbol literal consisting of a pound (or number) sign followed by a string. The examples above would look like this:

```
#'=='   #'abc'  #'negated' #'to:by:'  #'in:the:course:of:'
```

However, any character that is valid in a string can be in the extended symbol literal.

```
#' '   #'-----'   #'#'   #'A big step for mankind.'
```

An extended literal of the form:

```
#aSymbolString
```

produces a symbol with exactly the same value as the expression:

```
aSymbolString asSymbol
```

Frequently Asked Question

4.8 • What is the form of an array literal?

An array literal is formed from a pound (or number) sign and a pair of parentheses enclosing numeric literals, valid message selectors, character literals, string literals, and other array literals. (Not included are boolean literals and extended symbol literals.) Nested arrays need not have the leading pound sign.

`#( 2 3 4)`	An array of integers
`#('abc' 'def')`	An array of strings
`#(2 $c 7.3 'def' 2r1111)`	An array holding various objects
`#( #(1 2) #(3 4) #(5 6))`	An array of arrays
`#((1 2) (3 4) (5 6))`	The same array of arrays

IBM Smalltalk

IBM Smalltalk supports byte array literals which are formed from a pound (or number) sign and a pair of square brackets enclosing integer literals in the range 0-255.

`#[0 1 2 3 255 7]`	A byte array of 6 values

Variables and Names

Frequently Asked Question

4.9 • What characters can be used in names?

The following characters can be used in names.

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 _
```

Most implementations support an underscore character as if it were an alphabetic character. However, some support underscore as an alternate way of writing assignment; such usage is obsolete though.

See “Can I use an underscore character?” on page 59 for more information.

Names must start with an uppercase or lowercase letter (or an underscore where allowed).

Frequently Asked Question

4.10 • Are there conventions for naming classes?

Yes. Classes are virtually always named as one or more words in some natural language. The variable must start with an uppercase letter. By convention, each additional word starts with a capital letter and underscores are not used.

```
SortedCollection   LargeLandMammal   RetiredEmployee
```

Related sets of classes are often named with some common prefix or suffix.

```
Collection   SortedCollection   OrderedCollection
WriteFileStream   ReadFileStream   ReadWriteFileStream
```

Frequently Asked Question

4.11 • Are there conventions for naming methods?

Yes. Methods are virtually always named as one or more words in some natural language, or a standard term or abbreviation. By convention, each word but the first starts with a capital letter and underscores are not used.

```
sqrt    negated    min:    asFloat    employeeSpouseName    isEmpty
```

Methods with keyword selectors are named so that the part of the keyword selector that precedes a parameter names or implies that value.

```
on:from:to:    when:do:    translateBy:    truncateTo:    x:y:
indexOfSubcollection:startingAt:ifAbsent:
```

Frequently Asked Question

4.12 • What different kinds of variables are there?

There are ten kinds of variables in Smalltalk. Four kinds have some amount of global scope:

`Smalltalk` Global Variables

Global to all classes (in dictionary `Smalltalk`). See question *[4.22]*.

Class variables

Global to a class. See question *[4.25]*.

Pool variables

Global to classes that use the pool dictionary. See question *[4.29]*.

Special variables: `nil`, `true`, `false`, `self`, `super`, and `Smalltalk`

Global to all classes

The other six kinds are local variables. The scope of block parameters and locals depend on the implementation, with some older implementations not supporting local block variables, and making block parameters local to the method.

Instance variables

Specific to each instance

Class instance variables

Specific to class methods and subclasses. See question *[4.26]*.

Method parameters

Local to the method

Method local variables

Local to the method

Block parameters

Local to the block, except that some older implementations make block parameters local to the method.

Block local variables

Local to the block, except that some older implementations make block parameters local to the method.

See [4.48] *'Do blocks differ between implementations?'* on page 76.

In a method, instance variable names, method parameters and locals must be unique, and must not be the same as block parameters and locals. None can be the same as a special variable. Different blocks in the same method may have the same parameter and local variable names as other blocks in the method.

```
method: parm1 with: parm2
    | temp1 temp2 |
    temp1 := instVar1.
    self flareWith: [ :bparm |
        | btemp |
        btemp := pbarm + instVar1 ].
    self flareWith: [ :bparm2 |
        | btemp2 |
        btemp2 := pbarm2 + instVar1 ].
```

The variables `parm1`, `parm2`, `temp1`, `temp2`, and `instvar1` must all be unique, and must be different from `bparm`, `bparm2`, `btemp`, and `btemp2`. However, it would be valid to have `bparm2` be the same as `bparm`, and `btemp` the same as `btemp2`.

Frequently Asked Question

4.13 • What is the lifetime of these variables?

The list below shows the lifetime of each of the kinds of variables.

`Smalltalk` *Globals*

Forever; the variables can be deleted from the dictionary, thus shortening their 'lifetime'.

Class variables

The life of the class; the variables can be deleted from the class, thus shortening their 'lifetime'.

Pool variables

The life of the class or pool; the variables can be deleted from the pool, thus shortening their 'lifetime'.

Special variables

Forever.

Instance variables

The life of the instance.

Class instance variables

The life of the class; the variables can be deleted from the class, thus shortening their 'lifetime'.

Method parameters

The life of the method context.

Method local variables

The life of the method context.

Block parameters

The life of the block invocation, except for implementations not supporting block local variables, in which case it is the life of the method context.

Block local variables

The life of the block invocation, except for implementations not supporting block local variables, in which case it is the life of the method context.

Frequently Asked Question

4.14 • What is the search order for global variables?

Global variables are found by searching first for a class variable, then searching pool dictionary list in the order specified in the class definition, and then looking in the dictionary `Smalltalk`.

Frequently Asked Question

4.15 • Are there conventions for naming variables?

Instance variables are usually named either by the class of the data, or by the expected use or actual contents, or sometimes both. Skublics† calls these typed and semantic names.

Examples of typed names:

`aString`	An instance of `String`
`anInteger`	An instance of `Integer` or one of its subclasses
`aNumber`	An instance of `Number` or one of its subclasses

Examples of semantic names:

`numberOfBurners`	The number of burners (of a stove or hot air balloon)
`nameOfEmployee`	Some object holding an employee name

Examples of combined typed and semantic names:

`employeeNameString`	An instance of `String` holding an employee name
`burnerCountInteger`	An instance of `Integer` (or of a subclass)

Frequently Asked Question

4.16 • Are there conventions for naming instance variables?

Instance variables hold the data for classes and are defined in the class itself. Since there is little context from which to imply a meaning, the names should be descriptive of the purpose and use of the variable. Thus, instance variables should be named using semantic names.

`employeeName`	An employe name
`numberOfCylinders`	A count
`lotSizeInAcres`	A number of some kind, giving the lot size in acres

Frequently Asked Question

4.17 • Are there conventions for naming class variables?

Class variables hold various bits of data, and are defined in the class itself. Since there is little context from which to imply a meaning, the names should be

†. See *Smalltalk with Style* in question [1.7].

descriptive of the purpose and use of the variable. Thus, class variables should be named using semantic names.

`MaximumCylinderCount`	The most!
`NextEmployeeNumber`	What's to come next

See [4.25] *'What is a class variable?'* for more information.

Frequently Asked Question

4.18 • Are there conventions for naming method parameters?

Method parameters can be named using either typed or semantic naming, depending on context. Typed names are useful when a value is of some expected class and context provides semantic clues:

```
cylinders: anInteger
   anInteger > MaximumCylinderCount ifTrue: [ "error" ].
   numberOfCylinders := anInteger
```

Semantic names are useful when context either does not provide semantic clues or the code is clearer anyway. Compare:

```
taxOnBuildingLot: lotSizeInAcres
   ^ lotSizeInAcres * self buildingLotTaxRate
```

with:

```
taxOnBuildingLot: aNumber
   ^ aNumber * self buildingLotTaxRate
```

Frequently Asked Question

4.19 • Are there conventions for naming temporary variables?

Local variables are best if named with a meaningful semantic name that indicates how the variable is used.

Some authors recommend using a new variable and a new name for each different use, rather than naming variables with simple names like `n` or `temp` and reusing them.

```
(need example)
```

Example 6, Display

Frequently Asked Question

4.20 • When can variables have leading capital letters?

Leading upper-case letters are required on global variables (as defined in the dictionary `Smalltalk`), class variables, and pool variables. They are optional on other variables.

Some implementations make exceptions:

Implementation	Must be lowercase
Digitalk	Local variables, method & block parameters
IBM Smalltalk	(none)
ParcPlace VisualWorks	?
VMARK Enfin	?

Frequently Asked Question

4.21 • Can I use an underscore character?

Some implementations of Smalltalk allow an underscore character to be used in variable names. Some implementations still treat an underscore as an alternative for the assignment operation.

Implementation	Is Underscore Allowed in Variables
IBM Smalltalk	Yes; may be leading character; acts as lowercase
ParcPlace VisualWorks	Yes.
Digitalk	No
VMARK Enfin	?

Global Variables

Frequently Asked Question

4.22 • What is a global variable?

A global variable is a variable that is available to all classes, or, sometimes just to more than one class.

See [4.12] *'What different kinds of variables are there?'*.

Frequently Asked Question

4.23 • When should global variables be used?

Most experts say to never use a global variable! In Smalltalk, it is particularly bad to use variables defined in the dictionary `Smalltalk` because they are global to everything. Only classes and pool dictionaries should be global.

Note: Some software development tools provide for namespaces in Smalltalk which limit the visibility of global variables to sections of a class library.

Frequently Asked Question

4.24 • What alternatives are there to global variables?

Instead of a global variable, use a class method. Have it return the same value as the global variable would have held. Values returned by methods are hidden. The method can be overridden by subclasses and a different value might be answered.

Some authorities recommend using class methods to answer constant values rather than using class variables:

```
maximumCylinderCount
    ^ 12
```

Some authorities recommend using a new class to hold values like the next employee number. The class, say `EmployeeNumber`, would have one instance per sequence of employee numbers. The next employee number would be thus be an instance variable.

Also see [4.27] *'What good is an class instance variable?'* on page 61 for another alternative.

Frequently Asked Question

4.25 • What is a class variable?

A class variable is a special variable which is global to the instance and class methods of a class and all of its subclasses. Class variables must start with an uppercase letter and are a part of the definition of the class itself.

Class variables are used as alternatives to global variables when the needed scope is a class, its subclasses, and their instances.

Frequently Asked Question

4.26 • What is a class instance variable?

A class instance variable is an instance variable of the class itself. It belongs to the class and its subclasses. It cannot be seen by instances of the class. It follows the normal rules for instance variables and can have a leading uppercase letter only if generally allowed in the implementation.

Frequently Asked Question

4.27 • What good is an class instance variable?

Class instance variables are rarely used by applications. One possible use is to hold a value which is returned by a class method, rather than use a class variable to hold the same value. Instead of writing:

```
MySpecialValue
```

to access a class variable, you might write:

```
self class mySpecialValue
```

which invokes the class method `mySpecialValue` which answers the value of a class instance variable holding the same value. To set the value, you might write:

```
self class mySpecialValue: aValue
```

There are several advantages:

Information hiding

Instances of the class or of its subclasses cannot access the variable directly.

Overriding in subclasses

Subclasses can override the method, providing modified access or hiding access.

Pool Dictionaries

Frequently Asked Question

4.28 • What are pool dictionaries?

Pool dictionaries are dictionaries which are examined by the compiler to resolve names found in methods. Each class can define its own pool dictionaries. Pool dictionaries are not inherited.

Pool dictionaries are usually used to hold the names of constants, often achieving the same purpose as header files of `#DEFINE` statements as used in C and C++.

Pool dictionary *names* are, however, themselves global variables and are stored in the `Smalltalk` dictionary.

Example 7 shows a possible pool dictionary definition.

```
" Definition of DiddleZork pool dictionary "
| pool |
pool := Dictionary new.
pool at: #FlagBits put: 2r10001000.
pool at: #ZonkFlag put: 16rFFFF0000.
pool at: #DiddleIt put: 2r00000001.
Smalltalk at: DiddleZork put: dict
```

Example 7

If a class uses the pool dictionary `DiddleZork`, it might hold code like that in Example 8.

```
" Use of DiddleZork pool dictionary "
FlagBits bitOr: DiddleIt
" Answers: 89 "
```

Example 8

Pool dictionaries can be set as in Example 9, but it is bad practice. Pool dictionaries should only hold constant values.

```
" Setting values in DiddleZork pool dictionary "
FlagBits := 2r10001111
```

Example 9

Frequently Asked Question

4.29 • What pool dictionaries come with Smalltalk?

Each vendor provides a different set of pool dictionaries.

IBM Smalltalk

Pool Dictionary	Dictionary Contents
`CfsConstants`	Constants used by file system calls
`CgConstants`	Constants used by graphics calls
`CldtConstants`	Characters often used in printable strings
`CwConstants`	Constants used by widgets calls
`SystemExceptions`	Values used in exception handling

Digitalk V/Mac 2.0

Pool Dictionary	Dictionary Contents
`CharacterConstants`	Values of ASCII characters: `Tab`, `Space`, `Lf`, `Cr`, ...
`EventConstants`	Values like: `Button1`, `Button2`, `ShiftKey`, ...

ParcPlace VisualWorks 2.0

Pool Dictionary	Dictionary Contents
`IOConstants`	Values like: `CR` and `LF`
`TextConstants`	Values like: `CR, LF, Space, Tab, Ctrl`*n*

Frequently Asked Question

4.30 • Should pool dictionary names used prefixes?

Yes, although most vendors do not do this for their own pools. If you define a pool dictionary, use a meaningful prefix to assist in identifying the values when others see them in the code.

For example, if your are defining a pool dictionary which holds various limits on financial transactions, you might name the pool dictionary `FinancialLimits` and prefix each entry with `FinLim`:

Pool Constant	Description
`FinLimWireTransfer`	The largest allowed wire transfer
`FinLimATMDailyWithdrawal`	The largest ATM cash withdrawal for a single day
`FinLimATMMinimumIncrement`	The smallest bill carried in an ATM machine

4.31 • Are there alternatives to pool dictionaries?

Yes.

Class methods

If there are not a huge number of values, then class methods can substitute for pool dictionaries. If the values apply to just one class (and its subclasses) and there are few of them, then the methods might go directly in the class itself. If there are more than a few, consider making a new class just to hold the constants.

Instance variable with a dictionary value

Rather than making a pool dictionary, make the same dictionary but put it into an instance variable. Each instance can share the same dictionary, or might have slightly different dictionaries, depending on the needs of the application. Access to the dictionary is simply:

```
dict at: key
```

Class instance variable with a dictionary value

Rather than making a pool dictionary, make the same dictionary but put it into an class instance variable, and make a class method that answers the dictionary. There is just one dictionary for the class and it takes up no space in the instance. Access to the dictionary is simply:

```
self class dict at: key
```

Frequently Asked Question

4.32 • When should pool dictionaries be used?

Pool dictionaries should be only when these four criteria hold true:

When there are a lot of named values

If there are just a few values it is better to have a class method return the value.

When the values go together in a coherent way

If the values are not connected they have no business in a pool dictionary together. They should be either in multiple pools or not in pools at all.

When the values are constant

Do not use pool dictionaries for values that change. Use them only for constant values, and write some code somewhere that initializes the pool dictionary when the application or class is loaded. Pools that are changed are giant traps for the unwary and will rise up and bite at inopportune times.

When the dictionary can be used by more than one class

If the values in a dictionary belong to just one class they are better off in an instance variable or class variable, even if it means having to do an explicit `at:` to get the value.

Classes

Frequently Asked Question

4.33 • How can I initialize classes?

Classes are initialized by code in the class itself. By convention, a class method named `initialize` is used, and is run once, by manual invocation, when the class is defined. When making changes to the dictionary, it is easy to forget to run the method. Many programmers put a comment at the top of the method:

```
" Don't forget to re-initialize me when you make changes:
       ThisClassName initialize     "
```

and then select the expression and evaluate it after making a change.

IBM Smalltalk and ENVY

Some implementations of Smalltalk automatically invoke a class `initialize` method when the class is loaded from a fileout. IBM Smalltalk and systems with Envy do this; others may. The fileout is generated with a line for each class initialize method like this:

```
Clunker initialize !
```

When classes are loaded from an unload file (using Envy or IBM Smalltalk), the class is sent the message `loaded`. It can then initialize itself.

"ENVY/Developer has two ways of automatically initializing stuff on load: `loaded` and the `SubApplication>>toBeLoadedCode`. The `toBeLoadedCode` message is run *before* your app is loaded. It is primarily used for things like pool dictionaries which are required for code to be properly compiled or linked on load. Basically it is just an arbitrary string that you define to do whatever you want. On load, the string is fetched, compiled and run."

"One thing that you should watch out for is multiple initializations. It is quite common to call, from `MyApp class>>loaded`, the initialization method of several classes `MyApp` defines. For initialization methods that initialize a class variable, you only want to initialize it once but if you blindly send #initialize (or whatever) to subclasses which inherit it from the class with the classVar, you could be initializing the classVar many times. Worse, you could reinitialize it when you load some app which subclasses your class."

"I generally define two load initialize paths: `initializeOnLoad` and `initializeOnLoadOneTime`. The first I automatically send to all classes defined by the app. This is necessary to initialize class instance variables (i.e., you must initialize each class individually). The second I send only to defined classes which directly implement `initializeOnLoadOneTime`. So, my `loaded` method looks something like:"

```
loaded
   self defined do: [:aClass |
      aClass initializeOnLoad.
         (aClass class includesSelector: #initializeOnLoadOneTime)
            ifTrue: [aClass initializeOnLoadOneTime]]
```

—Jeff McAffer, jeff@is.s.u-tokyo.ac.jp

Frequently Asked Question

4.34 • How do I create new classes from within a method?

The details of this differ; peek inside your favorite browser for details. But, note that this operation can only be done in the development environment; most vendors don’t allow it in a packaged application.

Frequently Asked Question

4.35 • How do I add a method to a class from within a method?

The details of this differ; peek inside your favorite browser for details. But, note that this operation can only be done in the development environment; most vendors don't allow it in a packaged application.

Frequently Asked Question

4.36 • What is a Metaclass?

A metaclass is an instance of class `Metaclass` (or `MetaClass` in some systems). Each class is an instance of an instance of `Metaclass`.

See *12.9 'Where is the 'new' method (or: Why is 'new' THERE!)?'* for details.

Frequently Asked Question

4.37 • How should an object be checked for class membership?

It's a bad idea to check an instance to see what its class is. It often indicates some real problem with the design of a class when it seems necessary to check an instance for membership in some class.

If it is necessary, check for membership in a hierarchy by asking if an instance belongs to a class or one of its subclasses.

```
thisThing isKindOf: Integer
```

Why? Because new subclasses can appear and disappear. The instance you have today may turn into an instance of a subclass tomorrow. Integers are a prime example. There never are any instances of `Integer` around. There may be instances of `SmallInteger`, `LargePositiveInteger`, and `LargeNegativeInteger`, or instances of `SmallInteger` and `LargeInteger`, or some other set of subclasses of `Integer`. Further, just what values map to just what classes varies by implementation.

This is true of most well designed hierarchies; classes come and go. Basic structure is less apt to change.

What Variables Hold

Frequently Asked Question

4.38 • What do Smalltalk variables hold?

It is commonly said that Smalltalk has solved the ’pointer problem’ since Smalltalk appears to not have any pointers at all. Smalltalk has solved the ’pointer problem’, but it did it by making *everything* a pointer rather than eliminating them. Since everything is a pointer (and there are no pointer manipulation operations), Smalltalk does not have the exposures of, say, C or C++ to pointer abuse.

Every value in every variable in Smalltalk is a pointer to the value it represents†. However, rarely does this fact become visible in programs. Since all variables always hold pointers to objects, it is common to speak as if variables held the objects themselves. Rather than saying *the string referenced by* `name` it is common to say *the string in* `name`.

See [7.4] *’How big are SmallIntegers?’* on page 120.
See [5.6] *’When do pointers become visible?’* on page 88.

Frequently Asked Question

4.39 • What is a type?

Popular languages that most programmers are familiar with, such as C, C++, COBOL, FORTRAN, PL/I, and Pascal, have types. A variable, say `zot`, cannot be declared without specifying, explicitly, or implicitly by some default rules, what it 'holds'. This is done using language keywords as in C:

```
long int zot;
float flot;
```

The variable `zot` holds (or describes or references or whatever) some memory. On many machines this will be 32 bits of memory aligned in some way. Since the type is `long int`, the compiler will generate code that treats these bits as an integer.

†. This is true even of integers although a universally implemented optimization encodes the value of smaller integers within the pointer itself. Such encoded integers are instances of `SmallInteger`.

The variable `flot` holds (or describes or references or whatever) some memory. This will be 32 bits of memory aligned in some way. Since the 'type' is `float`, the compiler will generate code that treats these bits as a floating point number.

The bits are otherwise the same. There is no distinction between memory locations that hold integers of 32 bits length and floating point numbers of 32 bits length. (Using some well-defined constructions, C programmers can even access the same memory 'cell' with variables of various types; FORTRAN programmers can, too, even more easily, with equivalence declarations.)

The language has to know at compile time what type a variable is, and it generates code using that knowledge. There are no tags on the data saying that 'this is an integer' or 'this is a float'.

Types are a characteristic of variables which aid the compiler in producing proper code.

Frequently Asked Question

4.40 • Does Smalltalk have types?

See [4.39] *'What is a type?'* for an earlier question on the same topic and [4.41] *'Is there any disagreement about types in Smalltalk?'* on page 71 for other views.

> As a noun, in programming, *type* defines the nature of a variable -- for example, integer, real number, text character, floating-point number, and so on.
>
> *—Microsoft Press Computer Dictionary. 2nd Ed.*

> Smalltalk ... is untyped.
>
> *—Cook, Hill & Canning, "Inheritance is not Subtyping" in Theoretical Aspects of Object-Oriented Programming, MIT Press, page 516.*

Smalltalk has variables but no language keywords to specify what the variable holds. Variables are declared by listing their names:

```
| zot flot |
```

The variables `zot` and `flot` can 'hold' objects.

See [4.38] *'What do Smalltalk variables hold?'*

So what are the types of `zot` and `flot`? There aren't any. (One could just as well say that they have the type 'object', but does a language with one type have types?)

Now, consider this code:

```
| zot flot |
zot  := 23.
flot := 23.45.
```

What are the types of the variables `zot` and `flot`? Does `zot` now have the type '`Integer`' and flot the type '`Float`'? No. They didn't have a type before and they don't after assignment.

Assigning a value to a variable in Smalltalk does not change the type of the variable, at least without doing great damage to the popularly held view of what 'type' means. A variable may 'hold' an integer without being of type `Integer`.

Now, consider:

```
| zot flot |
zot  := 23.
flot := 23.45.
flot := flot asInteger
```

Has `flot` changed its type yet again? No!

Consider this code:

```
holdsNil: aCollection
   aCollection
      do: [ :element |
         element = nil
            ifTrue: [ ^ true ] ].
   ^ false
```

What is the type of `aCollection`? Does it change its 'type' every time that the method is called? In this method `aCollection` can be *any* object that responds to `do:` by invoking a block and passing one parameter to it. It doesn't have to be a collection of any kind.

Saying that all types derive from type `Object`, as some proponents of other languages do, misses the point entirely. There are no types to derive from. `Object` is no more a type than `Float`.

Smalltalk inheritance is not a type inheritance but an implementation sharing inheritance. A subclass is not a subtype but an implementation of a new object that shares some or all of the implementation of the parent class.

Smalltalk values are tagged. The variable `aCollection` is self describing. There is no need to specify that a variable that holds it holds a collection. The compiler can generate code without having any type information -- the only operations in Smalltalk are message sends and assigns.

Smalltalk compilers optimize generated code, sometimes making non-binding assumptions about what a variable might hold. See [12.1] *'How can Smalltalk be optimized?'*

Frequently Asked Question

4.41 • Is there any disagreement about types in Smalltalk?

There is some disagreement on this topic, to put it mildly; saying that it causes flame wars again puts it mildly.

Some claim that, while it is literally true that Smalltalk variables have no type declaration, classes are really the types in Smalltalk programs. This seems to redefine what the term *type* means from a characteristic of a variable to a characteristic of data. While this kind of redefinition happens across time with all languages, it is an especially dangerous thing to happen to technical terms which must have a precise definition in order to be useful at all.

Others argue that Smalltalk *should* have a type system, and that programmers should declare the types of variables. OK, but then the result is not Smalltalk, it is some new language similar to Smalltalk. It may be better, or it may not, but it's not Smalltalk.

Yet others argue that some new thing should be introduced to replace types. In one proposal, the type of a variable becomes a list of the classes whose instances the variable might at some time hold. Again, this really defines a new language similar to Smalltalk.

*(Need to get pointers into the literature.)*n

Blocks

Frequently Asked Question

4.42 • What are blocks?

Blocks are expressions or groups of expressions enclosed in square brackets:

```
[ 2 + 3 ]

[  a := 1.
   b := 2.
   c := a + b ]
```

A block is itself an expression; its value is the block itself, not the result of evaluating the contents.

Blocks resemble small subroutines:

- Blocks must be explicitly invoked. Example 10 answers 1:

```
| a |
a := 1.
[ a := a + 1 ].
^ a
" Answers: 1 "
```

Example 10, Display

The block in the third line is effectively ignored; its value is the block itself but nothing is done with the result. The block must be explicitly invoked, or evaluated, as in Example 11:

```
| a |
a := 1.
[ a := a + 1 ] value.
^ a
" Answers: 2 "
```

Example 11, Display

Example 11 answers 2.

- Blocks can have parameters. Parameters are described in a header in the block; each starts with a colon:

```
[ :parameter1 :parameter2 … |
   expression … ]
```

Parameterized blocks are invoked with one of the *value* messages: `value:`, `value:value:`, `value:value:value:`, or `valueWithArguments:`.

Example 12 answers 3:

```
| a |
a := 1.
[ :increment |
   a := a + increment ] value: 2.
^ a
" Answers: 3 "
```

Example 12, Display

- Blocks can have local variables (in most implementations, but see [4.43]). Local variables follow the parameter header and have the same format as local variables in a method: they are a list of names surrounded by vertical bars:

```
[ :parameter1 :parameter2 … |
   | localVariable1 localVariable2 … |
   expression … ]
```

Example 13 answers 3:

```
| a |
a := 1.
[ :increment |
   | temp |
   temp := a + increment.
   a := temp ] value: 2.
^ a
" Answers: 3 "
```

Example 13, Display

Frequently Asked Question

4.43 • Are blocks objects?

Yes. Blocks are full fledged objects. They can be assigned to variables, placed into objects, and passed as parameters. The sort block used by sorted collections is a good example of passing a block as a parameter.

```
SortedCollection sortBlock: [ :first :second | first < second ]
```

Sort blocks can be assigned to variables and used later:

```
descending := [ :first :second | first < second ].
ascending  := [ :first :second | first > second ].
...
SortedCollection sortBlock: ascending
```

Frequently Asked Question

4.44 • When are block objects created?

Block objects are created at the point in a program when they are invoked, passed as a parameter, or assigned to a variable. In some cases, the compiler can optimize away the need to actually create a block object. In other cases, the block object must be created.

In Example 14, the block refers to a method local variable, `a`. Changes to the variable are reflected in the block no matter when it is evaluated.

```
| a block |
a := 1.
block := [ a ].
a := 3.
block value
" Answers: 3 "
```

Example 14, Display

In Example 15, the outer block is passed the block local variable, `a`, and it answers another block which answers the value of the parameter. This binds the value of `x` to that of `a` at the time of the assignment.

```
| a block |
a := 1.
block := [ :x | [x] ] value: a.
a := 3.
block value
" Answers: 1 "
```

Example 15, Display

See also Example 18 in 4.48 Do blocks differ between implementations?

Frequently Asked Question

4.45 • When is it useful to put blocks in variables?

Many authorities think that blocks should rarely be put into variables. But there are times when it is a useful practice, especially in just those circumstances where a large switch statement in C is useful: given some integer value with a large number of possible values, choose some action that depends on the value.

A common case occurs when processing a stream of characters or processing an error return code from an operating system.

Frequently Asked Question

4.46 • Are blocks optimized?

Blocks can be optimized and most implementations do some degree of optimization of blocks.

In the simplest of cases, `ifTrue:` and `ifFalse:` are not sent, and the block's code is expanded inline. While this may require compiling code for two cases (when the receiver *is* and *is not* boolean), it is much faster in the common case.

Frequently Asked Question

4.47 • How about optimizations of blocks not simply executed inline?

Optimizations can be performed on blocks that will be passed as parameters or stored into variables. These optimizations can make a considerable difference in both execution time and memory use.

In the general case, a block has to know about the method context in which it was assigned or from which it was passed. The block might refer to parameters of the method, to local variables of the method, or to instance variables of the object, or it might contain a return statement that would cause a return from the method.

If a block contains no references to variables outside of itself and has no return statement, then the block object can be smaller and simpler. A block that references only instance variables might also have a block object that is simpler than the general case.

In order to allow an implementation to perform those optimizations it can perform, it is best to code blocks according to these guidelines:

Make all temporary variables local to the block

Do not define *outside* the block those temporary variables that will be used only *inside* the block. That is, write:

```
| x |
x := [ :parm |
   | local |
   ... local ... ]
```

instead of:

```
| x local |
x := [ :parm |
   ... local ... ]
```

Pass instance variables or block local variables instead of referring to them directly

Instead of referencing instance variables directly, pass their values as parameters. That is, write:

```
| x |
x := [ :parm |
   ... parm ... ].
x value: instVar
```

instead of:

```
| x |
x := [ ... instVar ... ].
x value
```

Avoid returns from blocks

It's generally a bad idea to have a return from a block that is stored or passed since there is no guarantee that the context in which the block object was created still exists. Such blocks cannot be optimized since they must carry along the whole context of the method invocation in the method where they are written.

Frequently Asked Question

4.48 • Do blocks differ between implementations?

Yes. In particular, most Digitalk versions of Smalltalk, and earlier versions of VisualWorks do not support local variables within blocks; all locals variables must be declared at the method level.

Worse, they consider block parameters as method-wide local variables.

For example, Example 16 may have two method-wide local variables (`counter` and `index`) or just one (`counter`), depending on which implementation is used:

```
| counter |
counter := 0.
(1 to: 20)
   do: [ :index |
      counter := counter + index ].
```

Example 16, Display

Example 17 may answer some integer (in this case probably `2`) when run on a Smalltalk with method-level local variables. The problem is that the parameters have the same names as the two declared local variables, and the values passed to the block when it is sorted by the `addAll:` replace the values explicitly assigned. Other systems should flag such uses as errors.

```
| a b |
a := #(9 1 6 4 8).
b := SortedCollection sortBlock: [ :a :b | a > b ].
b addAll: a.
^ a
```

Example 17, Display

Example 18 assigns blocks to `blockArray`.

```
| blockArray dataArray |
blockArray := Array new: 5.
dataArray := #( 'Apple' 'Orange' 'Grape' 'Lemon' 'Kiwi').
1 to: blockArray size do: [ :index |
   blockArray
      at: index
      put: [ dataArray at: index ] ].
^ (blockArray at: 2) value
```

Example 18, Display

The block:

```
[ dataArray at: index ]
```

has a reference to the variable `index`. If `index` is a method-level local variable, then it is the same variable for each block stored and the desired effect will not occur.

If `index` is a block-local variable, then it is a different variable each time the block is invoked, and a different value will be stored with the block; this is the desired situation.

The example ends with the return of the value of the second saved block.

When run on Digitalk Smalltalk/V-Mac 2.0, this code gets a walkback in the stored block because of an index out of bounds. The value of `index` is 6, indicating that it is one more than the last value it had in the loop.

When run on ParcPlace VisualWorks 2.0, this code answers the string `'Orange'`.

—Adapted from Ralph Johnson, Classic Smalltalk Bugs

Frequently Asked Question

4.49 • What is a closure?

A *closure* is a *closed expression*, or an expression which carries along the meanings of its free variables. Closure is not a Smalltalk term, but comes from language theory. In Smalltalk, blocks in implementations which have local state (local variables private to the block) are called closures. Blocks in implementations where there are no block local variables and/or where block parameters are local variables in the containing method are not closures.

Methods

Frequently Asked Question

4.50 • What are private methods?

Private methods are methods whose author marked them as private using some convention or another. The idea is that the author of the code will not himself use them from outside the class tree in which they are defined. The author of the code has hopes that his friends won't use them either.

Private methods are indicated different ways in different implementations:

Digitalk Smalltalk

Indicated by a comment at the front of the method.

(What happens with Team/Tools?)

IBM Smalltalk and systems with ENVY

A separate method type is maintained by the system. Browsers can show plublic, private, or both.

ParcPlace VisualWorks

Separate categories are created for private methods.

4.51 • Can the privacy of private methods be enforced?

No. Marking a method private is like telling your dog not to eat that steak you're leaving out on the floor. Maybe your dog is well trained...

There are ways to see if a private method is being used from outside its intended scope. The techniques vary slightly by implementation.

VisualWorks

```
thisContext sender receiver == self
   ifFalse: [^self error: 'Hey, I''m private!'].
self privateStuff
```

> “Due to the run time cost of ’realizing’ a context object that is normally cached inside the VM, this technique is too expensive for most use, but using conditional compilation techniques described in our Jan 96 TSR [The Smalltalk Report] column, you could arrange to do such privacy checks during testing, and re-compile them out for delivery.”

—Jan Steinman, Barbara Yates <barbara.bytesmiths@acm.org>

IBM Smalltalk

The heart of the technique is a one-liner which answers the caller of the current method.:

```
Processor activeProcess stackAtFrame: 1 offset: -3
```

If used from within a block the results are undefined; it usually will answer the method that invoked the block but that’s not the same as the method that invoked the method that contains the block.

It’s handy to package up such expressions, say by making a new method. The first bit of code below is a new method for `Object` which answers its callers caller. (Thus the ’`1`’ becomes a ’`2`’ indicating one level further up the stack.)

```
! Object methods !
sender
   ^ Processor activeProcess stackAtFrame: 2 offset: -3
```

Example 19, Filein

This method can be used to check for privacy:

```
myMethod
   self sender = self ifFalse: [ self error: 'blah' ].
   " ... do your stuff ... "
```

A more comprehensive method, again for `Object`, is a bit slower but is simpler to use. It performs the full check for privacy and issues a message if violated:

```
! Object methods !
isPrivateMethod
   | activeProcess sender sendersSender |
   activeProcess := Processor activeProcess.
   sender        := activeProcess stackAtFrame: 1 offset: -3.
   sendersSender := activeProcess stackAtFrame: 2 offset: -3.
   sender = sendersSender
      ifFalse: [ self error: 'My sender is not myself.' ]
```

Example 20, Filein

Use it this way:

```
myMethod
   self isPrivateMethod.
   " ... do your stuff ... "
```

Inheritance, Self and Super

Frequently Asked Question

4.52 • What is self?

The special variable `self` represents the object on behalf of which a method is being run. It may be an instance of the class in which the method is located or it may be an instance of a subclass.

The value of `self` is considered a hidden parameter in all message sends. For example, in the expression:

```
employee name: aString
```

two parameters are passed: first, the value in `employee` which will become `self`, and then the value in `aString`.

Messages sent to `self` are sent to the object that `self` represents.

4.53 • What is super?

The special variable `super` is another name for `self`. It has one special property: messages sent to `super` are bound to a method found by starting the search in the parent of the class in which the method is written.

In its simplest and most common case, `super` is used in user written class methods named `new`.

```
" The standard new method idiom "
new
    ^ super new
       initialize
```

Example 21

In Example 21, an instance needs to be initialized. The normal idiom overrides the new method, gets a new instance, sends the `initialize` message to it, and returns the new instance (assuming that `initialize` answers `self`, which it usually does).

The problem comes when getting that new instance. The expression `self new` cannot be used since it would recursively reinvoke the method. Sending `new` to `super` solves the problem by asking the parent to do the work. Since the parent presumably did it right before we overrode `new`, it will still do it right now.

See [4.56] *'When should super not be used?'*

4.54 • Why can't I pass super as a parameter?

The `value` of `super` is identical to the value of `self`. If `super` is passed as a parameter or assigned to a variable, the value passed or assigned is that of `self`. its special properties are lost.

Messages intended for the parent of the class must be sent directly to `super`, literally:

```
super aSelector
```

Why is the implementation this way? The alternative is to have two values, one for `self` and one for `super`, which refer to the same object. They would have to compare equal using = but not using ==, which would be a bizarre result. Besides, there is no practical need to have two such values.

Also see: [4.55] *'Why doesn't cascading with super work?'*

Frequently Asked Question

4.55 • Why doesn't cascading with super work?

The following two bits of code are equivalent:

```
super m1; m2
```

and:

```
super m1.
self m2
```

That is, it's defined in the Blue Book this way and that's the way it is. Anyone who knows a good reason why it must be this way is invited to communicate with the editor.

But see [4.56] *'When should super not be used?'*.

Frequently Asked Question

4.56 • When should super not be used?

The variable `super` should *only* be used when the message name is the same as the current method name. That is, it should always be true that `methodName` is the same as `messageName` in:

```
methodName
   super messageName
```

Messages sent to any other name should use `self` and not `super`. Using `super` bypasses method(s) that belong to `self` (or lower parents of `self`). If that is your intent, you need to carefully rethink what you are doing. Such code is ugly, tricky, hard to read, hard to maintain, and will rise up and zap you or a teammate later.

Note to implementers: Compilers really should warn about this case.

Example 22 is similar to code found by the author in a commercial product. The `fromFile:` method uses `super new` to bypass the redundant `buildTitle` in `new`.

```
new
   super new buildTitle
fromFile: aString
   super new
      fileName: aString;
      buildTitleFrom: aString
```

Example 22

The problem comes when subclassing this code and overriding `new`. The method `fromFile:` will then bypass the subclasses new method. Such bugs are no fun to find. The original code might have been written like this. (Bold in a method shows changes.)

```
new
   super new initialize
fromFile: aString
   self new
      fileName: aString
initialize
   fileName isNil
      ifFalse: [ self buildTitle ]
      ifTrue:  [ self buildTitleFrom: fileName ]
```

Frequently Asked Question

4.57 • What is the difference between self and yourself?

The word `self` is a reserved word in Smalltalk that refers to the object which is the receiver of a message.

The word `yourself` is a message name which can be sent to any object. In response, the object answers `self`. That is, `yourself` has the implementation:

```
yourself
   ^self
```

`yourself` is used in cascaded messages to assure that the value answered by the cascade is the receiver of the cascade. For example, the code:

```
oc := OrderedCollection new
   add: 'hello';
   add: 'there
```

Example 23, Display

assigns `'there'` to `oc` since the last `add:` returns '`there`'. However, the code:

```
oc := OrderedCollection new
   add: 'hello';
   add: 'there;
   yourself
```

Example 24, Display

assigns the new instance of `OrderedCollection` to `oc` since `yourself` always answers the receiver.

(Need to ref questions on add:.)

—David Buck, The Object People

Frequently Asked Question

4.58 • Are there any uses of yourself except in cascades?

“Cascaded messages account for virtually all of the cases where `yourself` is used. There are some other possible uses, but they are contrived. Suppose I maintained a dictionary of messages to send to different objects to get back a string to display on the screen. I could write:

```
stringFor: anObject
   ^anObject perform: (dictionary at: anObject class)
```

In the dictionary, I could have:

```
Integer->#printString
Client->#formattedPrintString
String->#yourself
...
```

I would think twice (maybe three times) about using such a technique. It does, however, demonstrate that it's possible to have another meaningful use of `yourself`.”

—David Buck, dbuck@ccs.carleton.ca

Other Questions

Ones not having found a proper home yet...

Frequently Asked Question

4.59 • What does subclassResponsibility mean, and what is it used for?

The message `subclassResponsibility` causes an error message to be issued that says something like:

```
'My subclass should have implemented this message.'
```

It is used when a superclass needs to define some protocol but cannot provide an implementation. It documents an interface indicating that the method exists and it serves as documentation to implementers of subclasses that the method must be implemented. It is never intended that it be executed.

An example is `<` (less-than) in `Magnitude`. This message must be provided by all subclasses; a meaningful less-than comparison is what makes a magnitude be a magnitude. However, the actual implementation is very dependent on the data formats of the subclasses; thus `Magnitude` cannot do anything better than issue an error message.