Squeak Smalltalk: Language Reference

Version 0.0, 20 November 1999, by Andrew C. Greenberg, werdna@mucow.com
Version 1.2, 26 April 2001, by Andrew P. Black, black@cse.ogi.edu

Based on:
Smalltalk-80: The Language and Its Implementation, Author: Adele Goldberg and David Robson
Squeak Source Code, and the readers of the Squeak mailing list.

Squeak site: http://www.squeak.org

Contents

- Using Squeak
- Squeak Smalltalk Syntax

See also the Squeak Classes Reference page

Using Squeak: the Basics

- Mousing Around
- System Menus
- System Key Bindings

Mousing Around

Squeak (and the Smalltalk-80 from which it was spawned) assumes a machine with a three-button mouse (or its equivalent). These buttons were referred to as "red," "yellow" and "blue." The red button was conventionally used for selecting "things," the yellow button was conventionally used for manipulating "things" within a window and the blue button was conventionally used for manipulating windows themselves. Conventions, of course, are not always followed.

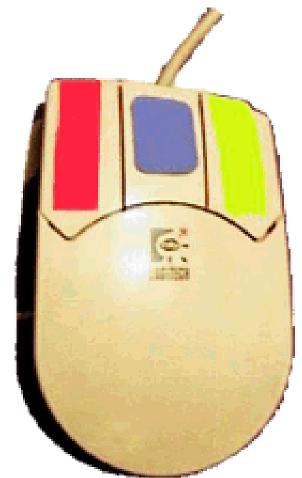
Since many modern mice no longer have three buttons, let alone colored ones, various mapping conventions are used:

For uncolored three-button mice, the usual mapping is:

left-mouse	->	red
middle-mouse	->	yellow
right-mouse	->	blue

Windows machines with three button mice can be made to conform to this mapping by right clicking the Windows title bar of Squeak, and selecting *VM Preferences >> Use 3 button mouse mapping.* Otherwise, for Windows machines with two button mice, the mapping is:

left-mouse	->	red
right-mouse	->	yellow
alt-left-mouse	->	blue



MacOS Systems generally have one mouse button. The mapping is:

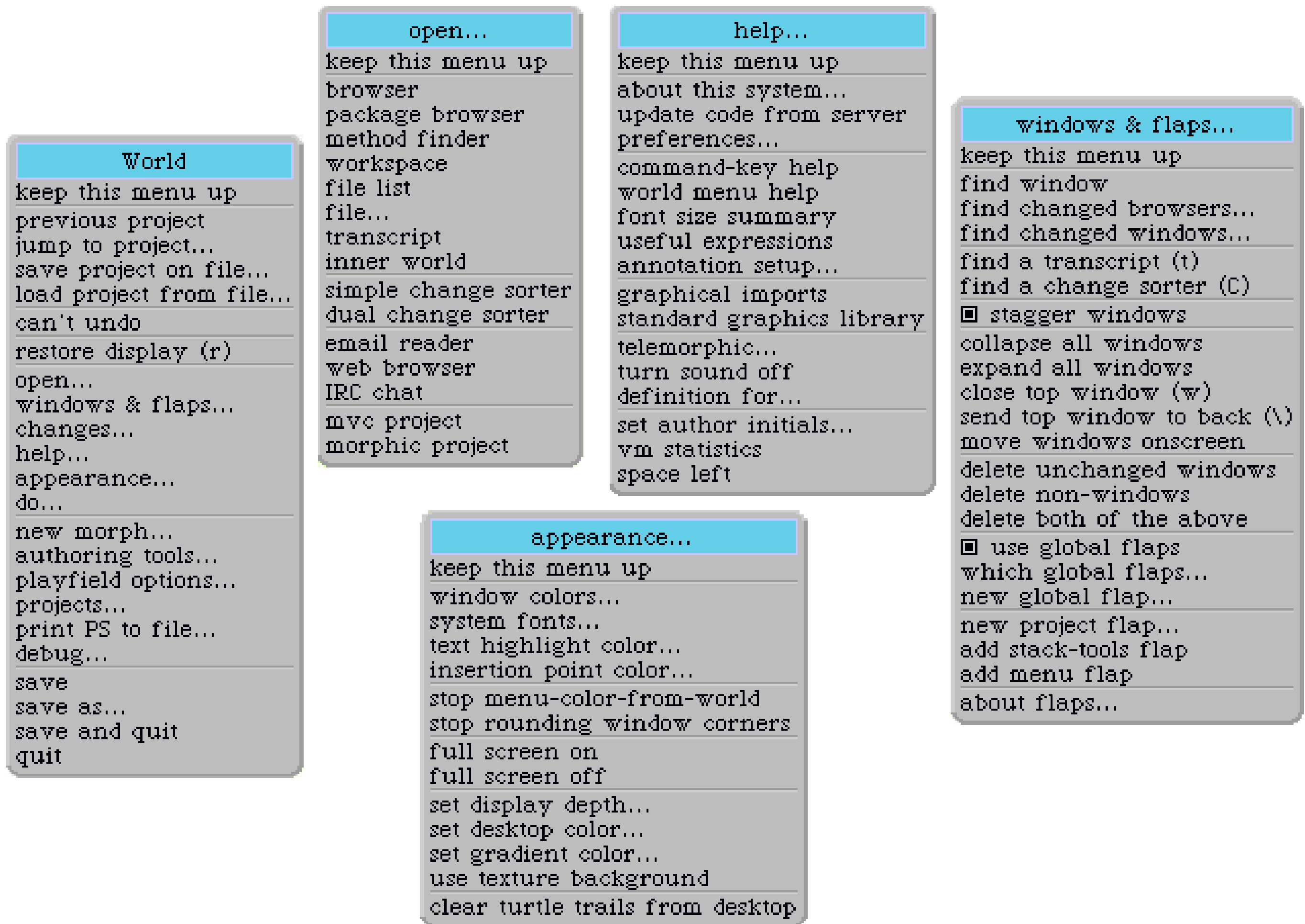
mouse	->	red
option-mouse	->	yellow
⌘-mouse	->	blue

It is worthwhile purchasing a 3-button mouse for your computer. I put colored labels on my buttons while I was training my fingers. I also chose a different mapping.

If you have a mouse with a scrolling wheel, map "wheel up" to command-upArrow and "wheel down" to command-downArrow, and you will be able to use the wheel to control scrolling in Squeak.

System Menus

Squeak provides access to certain Smalltalk services through its system menus, some of which are depicted below:



The Main Menu. The World menu, sometimes called the "main menu," can be reached by clicking the red button while the mouse points to the background of a system project. From this menu, you can save the image and changes files, save them in files with a different name, and terminate execution of the Squeak virtual machine. You can also access many other menus ... including the four shown here.

The open Menu provides access to many system tools, including system browsers, workspaces, change sorters, transcripts and file lists, as well as end user tools such as an email agent (*Celeste*) and web browser (*Scamper*).

The help Menu provides access to certain on-line help facilities as well as a preferences dialog, some environment enquiries, a dictionary and facilities for updating your version of Squeak.

The windows and flaps Menu provides access to services for manipulating system windows and (in Morphic only) *flaps*. Flaps are small tabs at the side of the screen that pull out like drawers and provide quick access to whatever you place there. Try them! The *Tools* flap is a very convenient way of getting new system tools (rather than using the *open* menu).

The appearance menu lets the user change various aspects of the systems appearance. In particular, it provides a way of adjusting the display depth and going in and out of full screen mode.

System Key Bindings

Applications that use standard Squeak text container widgets, including System Browsers, Workspaces, File Lists and Transcripts, provide facilities for manipulating the text and providing access to other system functionality. Many of these facilities can be reached by using the red-button menus, but many are more conveniently accessed using special key sequences. Of course, particular applications can use some, all or of none of these. In the following tables, a lower-case or numeric command "key" can be typed by simultaneously pressing the key and the Alt key (on Windows) or the ⌘ key (on MacOS). Upper-case keys are typed by simultaneously pressing either Shift-Alt (or Shift-⌘) and the indicated key, *or* ctrl and the indicated key. Other special key presses are indicated below in square brackets.

General Editing Commands

Key	Description	Notes
z	Undo	
x	Cut	
c	Copy	
v	Paste	
a	Select all	
D	Duplicate. Paste the current selection over the prior selection, if it is non-overlapping and legal	1
e	Exchange. Exchange the contents of current selection with the contents of the prior selection	1
y	Swap. If there is no selection, swap the characters on either side of the insertion cursor, and advance the cursor. If the selection has 2 characters, swap them, and advance the cursor.	
w	Delete preceding word	

Notes

1. These commands are a bit unusual: they concern and affect not only the current selection, but also the immediately preceding selection.

Search and Replace

Key	Description	Notes
f	Find. Set the search string from a string entered in a dialog. Then, advance the cursor to the next occurrence of the search string.	
g	Find again. Advance the cursor to the next occurrence of the search string.	
h	Set Search String from the selection.	
j	Replace the next occurrence of the search string with the last replacement made	
A	Advance argument. Advance the cursor to the next keyword argument, or to the end of string if no keyword arguments remain.	
J	Replace all occurrences of the search string with the last replacement made	
S	Replace all occurrences of the search string with the present change text	

Cancel/Accept

Key	Description	Notes
l	Cancel (also "revert"). Cancel all edits made since the pane was opened or since the last save	
s	Accept (also "save"). Save the changes made in the current pane.	
o	Spawn. Open a new window containing the present contents of this pane, and then reset this window to its last saved state (that is, cancel the present window).	

Browsing and Inspecting

Key	Description	Notes
b	Browse "it" (where "it" is a class name). Opens a new browser.	1
d	Do "it" (where "it" is a Squeak expression)	1
i	Inspect "it": evaluate "it" and open an inspector on the result. ("it" is a Squeak expression). Exception: in a method list pane, i opens an inheritance browser.	1
m	Open a browser of methods implementing "it" (where "it" is a message selector)	1,2
n	Open a browser of methods that send "it" (where "it" is a message selector).	1,2
p	Print "it". Evaluate "it" and insert the results immediately after "it." (where "it" is a Smalltalk expression)	1
B	Set the *present* browser to browse "it" (where "it" is a class name)	1
E	Open a browser of methods whose source contain strings with "it" as a substring.	1
I	Open the Object Explorer on "it" (where "it" is an expression)	1
N	Open a browser of methods using "it" (where "it" is an identifier or class name)	1
O	Open single-message browser (in selector lists)	1
W	Open a browser of methods whose selectors include "it" as a substring.	1

Notes:

1. A null selection will be expanded to a word, or to the whole of the current line, in an attempt to do what you want.
2. For these operations, "it" means the *outermost* keyword selector in a large selection.

Special Conversions and Processing

Key	Description	Notes
C	Open a workspace showing a comparison of the selection with the contents of the clipboard	
U	Convert linefeeds to carriage returns in selection	
X	Force selection to lowercase	
Y	Force selection to uppercase	
Z	Capitalize all words in selection	

Smalltalk Program Data Entry

Key	Description	Notes
q	Attempt to complete the selection with a valid and defined Smalltalk selector. Repeated commands yield additional selectors.	
r	Recognizer. Invoke the Squeak glyph character recognizer. (Terminate recognition by mousing out of the window)	
F	Insert 'ifFalse:'	
T	Insert 'ifTrue:'	
V	Paste author's initials, date and time.	
L	Outdent (move selection or line one tab-stop left)	
R	Indent (move selection or line one tab stop right)	
[Ctl-return]	Insert return followed by as many tabs as the previous line (with a further adjustment for additional brackets in that line)	
[shift-delete]	Forward delete. Or, deletes from the insertion point to the beginning of the current word.	

Bracket Keys

These keys are used to enclose (or un-enclose, if the selection is already enclosed) the selection in a kind of "bracket". Conveniently, double clicking just inside any bracketed text selects the entire text, but not the brackets.

Key	Description	Notes
Control-(	Enclose within (and), or remove enclosing (and)	
Control- [	Enclose within [and], or remove enclosing [and]	
Control- {	Enclose within { and }, or remove enclosing { and }	
Control- <	Enclose within < and >, or remove enclosing < and >	
Control- '	Enclose within ' and ', or remove enclosing ' and '	
Control- "	Enclose within " and ", or remove enclosing " and "	

Keys for Changing Text Style and Emphasis

Key	Description	Notes
k	Set font	
u	Align	
K	Set style	
1	10 point font	
2	12 point font	
3	18 point font	
4	24 point font	
5	36 point font	
6	Brings up a menu, providing choice of color, action-on-click, link to class comment, link to method, url.. To remove these properties, select more than the active part and then use command-0.	
7	bold	
8	italic	
9	narrow (same as negative kern)	
0	plain text (removes all emphasis)	
- (minus)	underlined (toggles it)	
=	struck out (toggles it)	
_ (a.k.a. shift -)	negative kern (letters 1 pixel closer)	
+ (a.k.a. shift =)	positive kern (letters 1 pixel further apart)	

Squeak Smalltalk Syntax: the Basics

- Pseudo-variables
- Identifiers
- Comments
- Literals
- Assignments
- Messages
- Expression Sequences
- Cascades
- Expression Blocks
- Control Structures
- Brace Arrays
- Class Definition
- Method Definition

Pseudo-Variables

Pseudo-variable	Description
nil	The singleton instance of Class UndefinedObject
true	The singleton instance of Class True
false	The singleton instance of Class False
self	The current object, that is, the receiver of the current message.
super	As the receiver of a message, **super** refers to the same object as **self**. However, when a message is sent to **super**, the search for a suitable method starts in the superclass of the class whose method definition contains the word **super**.
thisContext	The active context, that is, the "currently executing" MethodContext or BlockContext.

- Pseudo-variables are reserved identifiers that are similar to keywords in other languages.
- **nil**, **true** and **false** are constants.
- **self**, **super** and **thisContext** vary dynamically as code is executed.
- It is not possible to assign to any of these pseudo-variables.

Identifiers

letter (letter | digit)*

- Smalltalk identifiers (and symbols) are **case-sensitive**.
- It is a Smalltalk convention for identifiers (instance and temporaries) of several words to begin with a lower case character, and then capitalize subsequent words. (*e.g.*, thisIsACompoundIdentifier).
- Certain identifiers, for example, globals (*e.g.* , Smalltalk) and class variables, are by convention initially capitalized. The names of all classes are also global variables (*e.g.*, SystemDictionary).

Comments

"a comment comprises any sequence of characters, surrounded by double quotes"
"comments can include the 'string delimiting' character"
"and comments can include embedded double quote characters by ""doubling"" them"
"comments can span many
many
lines"

Literals (Constant Expressions)

Numbers (Instances of class Number)

In the following, ==> means "prints as".

Decimal integer: **1234**, **12345678901234567890**
Octal integer: **8r177**, **8r1777777777777777777777**
Hex integer: **16rFF**, **16r123456789ABCDEF012345**
Arbitrary base integer: **2r1010** ==> 10
Integer with exponent: **123e2** ==> 12300, **2r1010e2** ==> 40
Float (double precision): **3.14e-10**
Arbitrary base float: **2r1.1** ==> 1.5
Float with exponent: **2r1.1e2** ==> 6.0

- Squeak supports SmallInteger arithmetic (integers between -2^{30} and 2^{30-1}) with fast internal primitives.
- Squeak supports arbitrary precision arithmetic seamlessly (automatically coercing SmallInteger to LargePositiveInteger and LargeNegativeInteger where appropriate), albeit at a slight cost in speed.
- Squeak supports several other kinds of "numeric" value, such as Fractions (arbitrary precision rational numbers) and Points. While there are no literals for these objects, they are naturally expressed as operations on built-in literals. ("2/3" and "2@3", respectively)
- Numbers may be represented in many radices, but the radix specification itself is always expressed in base 10. The base for the exponent part is the same as the radix. So: **2r1010** ==> 10, **10e2** ==> 1000 (=10 x 10^2), but **2r1010e2** ==> 40 (=10 x 2^2)

Characters (Instances of class Character)

$x "A character is any character (even unprintable ones), preceded by a dollar sign"
$3 "Don't be shy about characters that are digits"
$< "or symbols"
$$ "or even the dollar sign"

Strings (Instances of class String)

'a string comprises any sequence of characters, surrounded by single quotes**'**
'strings can include the "comment delimiting" character**'**
'and strings can include embedded single quote characters by doubling'' them**'**
'strings can contain embedded
newline characters**'**
'' "and don't forget the empty string"

- A string is very much like ("isomorphic to") an array containing characters. Indexing a string answers characters at the corresponding position, staring with 1.

Symbols (Instances of class Symbol)

#**'**A string preceded by a hash sign is a Symbol**'**
#orAnyIdentifierPrefixedWithAHashSign
#orAnIdentifierEndingWithAColon:
#or:several:identifiers:each:ending:with:a:colon:
#- "A symbol can also be a hash followed by '-' or any special character"
#+< "or a hash followed by any pair of special characters"

- Symbol is a subclass of String, and understands, in large part, the same messages.
- The primary difference between a symbol and a string is that all symbols comprising the same sequence of characters are the same instance. Two different string instances can both have the characters 'test one two three', but every symbol having the characters #'test one two three' is the same instance. This "unique instance" property means that Symbols can be efficiently compared, because equality (=) is the same as identity (==).
- "Identifier with colon" Symbols (*e.g.,* #a:keyword:selector:) are often referred to as keyword selectors, for reasons that will be made clear later.
- "Single or dual symbol" Symbols (*e.g.,* #* or #++) are often referred to as binary selectors.
- The following are permissible special characters: +/*\~<=>@%|&?!
- Note that #-- is not a symbol (or a binary selector). On the other hand, #'--' *is* a symbol (but not a binary selector).

Constant Arrays (Instances of class Array)

#(1 2 3 4 5 **)** "An array of size 5 comprising five Integers (1 to 5)"
#('this' #is $a #'constant' array **)** "An array of size 5 comprising a String ('this'), a Symbol (#is), a Character ($a) and two Symbols (#constant and #array)."
#(1 2 (1 #(2) 3) 4 **)** "An array of size 4 comprising two Integers (1 and 2), an Array of size 3, and another Integer (4)."
#(1 + 2 **)** "An array of size 3 comprising 1, #+, and 2. It is *not* the singleton array comprising 3."

- Constant arrays are constants, and their elements must therefore be constants. "Expressions" are not evaluated, but are generally parsed as sequences of symbols as in the example above.
- Constant arrays may contain constant arrays. The hash sign for internal constant arrays is optional.
- Identifiers and sequences of characters in constant arrays are treated as symbols; the hash sign for internal symbols is optional.
- Arrays are indexed with the first element at index 1.

Assignments

identifier ← expression
identifier := expression " := is always a legal alternative to ←, but the pretty printer uses ← "

foo ← 100 factorial
foo ← bar ← 1000 factorial

- The identifier (whether instance variable, class variable, temporary variable, or otherwise) will thereafter refer to the object answered by the expression.
- The " ← " glyph can be typed in Squeak by keying the underbar character (shift-hyphen).
- Assignments are expressions; they answer the result of evaluating the right-hand-side.
- Assignments can be cascaded as indicated above, resulting in the assignment of the same right-hand-side result to each variable.

Messages

Unary Messages

theta sin
quantity sqrt
nameString size
1.5 tan rounded asString "same result as (((1.5 tan) rounded) asString)"

- Unary messages are messages without arguments.
- Unary messages are the most "tightly parsed" messages, and are parsed left to right. Hence, the last example answers the result of sending #asString to the result of sending #rounded to the result of sending #tan to 1.5

Binary Messages

3 + 4 " ==> 7 "
3 + 4 * 5 " ==> 35 (**not** 23) "
3 + 4 factorial " ==> 27 (**not** 5040) "
total - 1
total <= max "true if total is less than or equal to max"
(4/3)*3 = 4 "==> true — equality is just a binary message, and Fractions are exact"
(3/4) == (3/4) "==> false — two equal Fractions, but not the same object"

- Binary messages have a receiver, the left hand side, and a single argument, the right hand side. The first expression above sends 3 the message comprising the selector #+ with the argument 4.
- Binary messages are *always* parsed left to right, without regard to precedence of numeric operators, unless corrected with parentheses.
- Unary messages bind more tightly than binary messages

Keyword Messages

12 between: 8 and: 15 " ==> true "
#($t $e $s $t) at: 3 " ==> $s "
array at: index put: value "==> answers value, after storing value in array at index"
array at: index factorial put: value "same, but this time stores at index factorial"
1 to: 3 do: aBlock "This sends #to:do: (with two parameters) to integer 1"
(1 to: 3) do: aBlock "This sends #do: (with one parameter) to the Interval given by evaluating '1 to: 3'"

- Keyword messages have 1 or more arguments
- Keyword messages are the least-tightly binding messages. Binary and unary messages are resolved first unless corrected with parentheses.

Expression Sequences

expressionSequence ::= expression (. expression)* (.)^{opt}

```
box ← 20@30 corner: 60@90.
box containsPoint: 40@50
```

- Expressions separated by *periods* are executed in sequence.
- Value of the sequence is the value of the final expression.
- The values of all of the other expressions are ignored.
- A final period is optional.

Cascade Expressions

```
receiver
    unaryMessage;
    + 23;
    at: 23 put: value;
    yourself
```

- messages in a cascade are separated by *semicolons*; each message is sent to `receiver` in sequence.
- Intermediate answers are ignored, but side-effects on `receiver` will be retained.
- The cascade answers the result of sending the last message to `receiver` (after sending all the preceding ones!)

Block Expressions

Blocks, actually instances of the class BlockContext. They are used all the time to build control structures. Blocks are created using the **[ ]** syntax around a sequence of expressions.

> **[** expressionSequence **]** "block without arguments"
> **[** (: identifier)$^+$ | expressionSequence **]** "block with arguments"
> **[** (: identifier)$^+$ | | identifier$^+$ | expressionSequence **]** "block with arguments and local variables"

[1. 2. 3 **]** "a block which, when evaluated, will answer the value 3"
[object doWithSideEffects. test**]** "a block which, when evaluated, will send #doWithSideEffects to object, and answer the object test"
[:param **|** param doSomething **]** "a block which, when evaluated with a parameter, will answer the result of sending #doSomething to the parameter.

- A block represents a deferred sequence of actions.
- The value of a block expression is an object that can execute the enclosed expressions at a later time, if requested to do so. Thus
  - **[** 1. 2. 3 **]** ==> [ ] in UndefinedObject>>DoIt
  - **[** 1. 2. 3 **]** value ==> 3
- Language experts will note that blocks are rougly equivalent to lambda-expressions, anonymous functions, or closures.

Evaluation Messages for BlockContext

Message	Description	Notes
value	Evaluate the block represented by the receiver and answer the result.	1
value: arg	Evaluate the block represented by the receiver, passing it the value of the argument, arg.	2
valueWithArguments: anArray	Evaluate the block represented by the receiver. The argument is an Array whose elements are the arguments for the block. Signal an error if the length of the Array is not the same as the the number of arguments that the block was expecting.	3

Notes

1. The message `#value`, sent to a block, causes the block to be executed and answers the result. The block must require zero arguments.
2. The message `#value: arg`, causes the block to be executed. The block must require exactly one argument; the corresponding parameter is initialized to `arg`.
3. Squeak also recognizes `#value:value:`, `#value:value:value:` and `#value:value:value:value:`. If you have a block with more than four parameters, you must use `#valueWithArguments:`

Control Structures

Alternative Control Structures (Receiver is Boolean)

Message	Description	Notes
ifTrue: alternativeBlock	Answer nil if the receiver is false. Signal an Error if the receiver is nonBoolean. Otherwise, answer the result of evaluating alternativeBlock	1,2
ifFalse: alternativeBlock	Answer nil if the receiver is true. Signal an Error if the receiver is nonBoolean. Otherwise answer the result of evaluating the argument, alternativeBlock.	1,2
ifTrue: trueAlternativeBlock **ifFalse:** falseAlternativeBlock	Answer the value of trueAlternativeBlock if the receiver is true. Answer the value of falseAlternativeBlock if the receiver is false. Otherwise, signal an Error.	1,2
ifFalse: falseAlternativeBlock **ifTrue:** trueAlternativeBlock	Same as ifTrue:ifFalse:.	1,2

Notes

1. These are not technically control structures, since they can be understood as keyword messages that are sent to boolean objects. (See the definitions of these methods in classes True and False, respectively).
2. However, these expressions play the same role as control structures in other languages.

Alternative Control Structures (Receiver is any Object)

Message	Description	Notes
ifNil: nilBlock	Answer the result of evaluating nilblock if the receiver is nil. Otherwise answer the receiver.	
ifNotNil: ifNotNilBlock	Answer the result of evaluating ifNotNilBlock if the receiver is not nil. Otherwise answer nil.	
ifNil: nilBlock **ifNotNil:** ifNotNilBlock	Answer the result of evaluating nilBlock if the receiver is nil. Otherwise answer the result of evaluating ifNotNilBlock.	
ifNotNil: ifNotNilBlock **ifNil:** nilBlock	Same as #ifNil:ifNotNil:	

Iterative Control Structures (receiver is aBlockContext)

Message	Description	Notes
whileTrue	Evaluate the receiver. Continue to evaluate the receiver for so long as the result is true.	
whileTrue: aBlock	Evaluate the receiver. If true, evaluate aBlock and repeat.	
whileFalse	Evaluate the receiver. Continue to evaluate the receiver for so long as the result is false.	
whileFalse: aBlock	Evaluate the receiver. If false, evaluate aBlock and repeat.	

Enumeration Control Structures (Receiver is anInteger)

Message	Description	Notes
timesRepeat: aBlock	Evaluate the argument, aBlock, the number of times represented by the receiver.	
to: stop **do:** aBlock	Evaluate aBlock with each element of the interval (self to: stop by: 1) as the argument.	
to: stop **by:** step **do:** aBlock	Evaluate aBlock with each element of the interval (self to: stop by: step) as the argument.	

Enumeration Control Structures (Receiver is Collection)

Message	Description	Notes
do: aBlock	For each element of the receiver, evaluate aBlock with that element as the argument.	1

Note

1. Squeak collections provide a very substantial set of enumeration operators. See the section Enumerating Collections on the Classes Reference.

Case Structures (Receiver is any Object)

Message	Description	Notes
caseOf: aBlockAssociationCollection	Answer the evaluated value of the first association in aBlockAssociationCollection whose evaluated key equals the receiver. If no match is found, signal an Error.	1
caseOf: aBlockAssociationCollection **otherwise:** aBlock	Answer the evaluated value of the first association in aBlockAssociationCollection whose evaluated key equals the receiver. If no match is found, answer the result of evaluating aBlock.	1

Note

1. aBlockAssociationCollection is a collection of Associations (key/value pairs).
   Example: `aSymbol caseOf: {[#a]->[1+1]. ['b' asSymbol]->[2+2]. [#c]->[3+3]}`

Expression "Brace" Arrays

braceArray ::= { expressionSequence }

{ 1. 2. 3. 4. 5 } "An array of size 5 comprising five Integers (1 to 5)"
{ $a #brace array } "An array of size 3 comprising a Character ($a) a Symbol (#brace), and *the present value of* the variable array."
{ 1 + 2 } "An array of size 1 comprising the single integer 3."

- Brace arrays are bona-fide Smalltalk expressions that are computed at runtime.
- The elements of a brace array are the answers of its component expressions.
- They are a sometimes convenient and more general alternative to the clunky expression "Array with: expr1 with: expr2 with: expr3"
- Indexing is 1-based.

Answer Expressions

answerExpression ::= ^ expression

^ aTemporary
^ 2+3

- Inside the body of a method, an answer expression is used to terminate the execution of the method and deliver the expression as the method's answer.
- Answer expressions inside a nested block expression will terminate the enclosing method.

Class Definition

Ordinary Class Definition

```
SuperClass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Major-Minor'
```

Variable Class Definition

These forms of class definition are used to create indexable objects, *i.e.*, those like Array, ByteArray and WordArray. They are included here for completeness, but are not normally used directly; instead, use an ordinary object with an instance variable whose value is an approriate Array (or other collection) object.

```
SuperClass variableSubclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Major-Minor'

SuperClass variableByteSubclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Major-Minor'

SuperClass variableWordSubclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Major-Minor'
```

Method Definition

All methods answer a value; there is an implicit `^ self` at the end of every method to make sure that this is the case. Here is an example (from class String).

```
lineCount
     "Answer the number of lines represented by the receiver, where every
     cr adds one line."

    | cr count |
    cr ← Character cr.
    count ← 1  min: self size.
    self do:
        [:c | c == cr ifTrue: [count ← count + 1]].
    ^ count
```