Squeak Smalltalk: Quick Reference

Version 0.0, 20 November 1999, by Andrew C. Greenberg, werdna@mucow.com
Version 01., 10 April 2001, by Andrew P. Black, black@cse.ogi.edu
PDF Conversion by Russell Allen, russell.allen@firebirdmedia.com

Based on:

Smalltalk-80: The Language and Its Implementation, Author: Adele Goldberg and David Robson
Squeak Source Code v. 2.6
and the readers of the Squeak mailing list.

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

This document is in the Squeak Documentation Archive at: http://docs.squeakspace.com/quickreference

Table of Contents

1. Squeak Smalltalk: Language Reference

1.1. Using Squeak: the Basics

1.1.1. 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 view or window and the blue button was conventionally used for manipulating windows themselves. Conventions, of course, are not always followed and your mileage may vary.

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

For uncolored three-button mice, the mapping is:

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

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

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

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

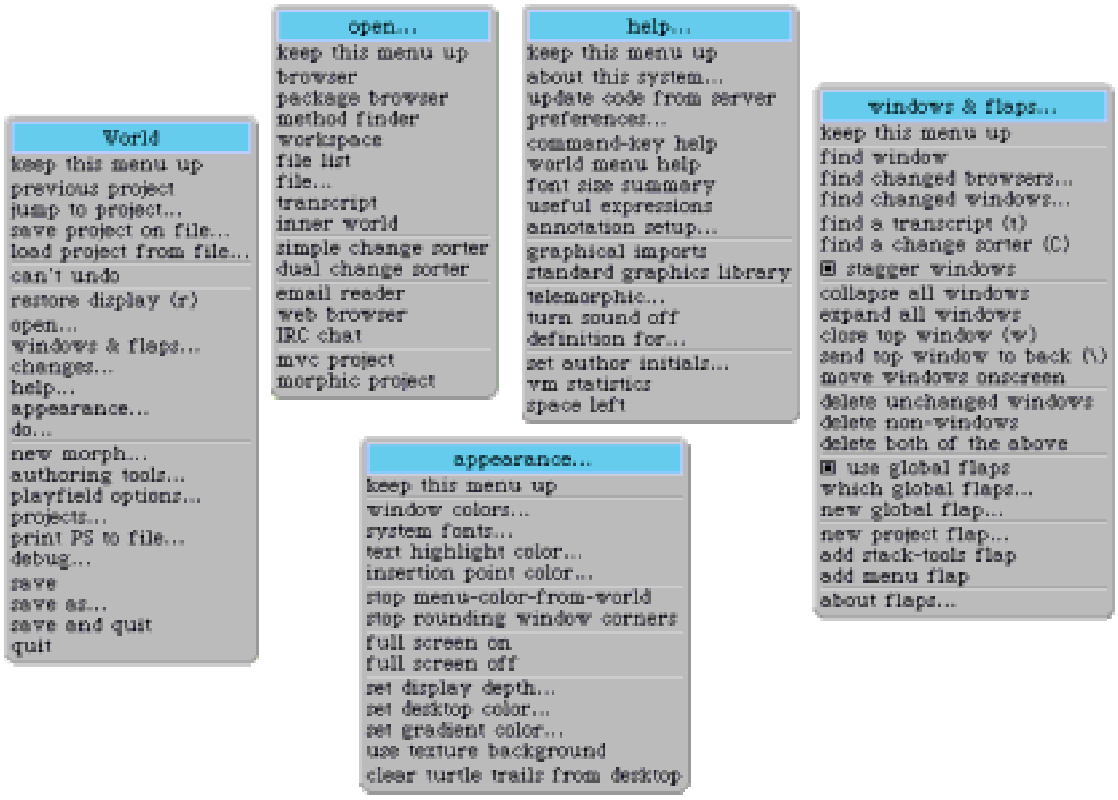
```
mouse           -> red
option-mouse    -> yellow
cmd-mouse       -> blue
```

If you purchase a 3-button mouse for your computer, you will be pleased that you did so! I put colored sticky labels on my buttons when I was first training my fingers.

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

1.1.2. 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 three 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.

1.1.3. 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 contents of the immediately previous selection	1
y	Swap. If the selection has 0 characters, swap the immediately surrounding characters. If the selection has 2 characters, swap them.	
w	Delete preceding word	

Notes

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 interitance 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:

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.

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 unenclose if the selection is already enclosed) the selection in a kind of "bracket". As an aside, 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 "	

Special 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	color, action-on-click, link to class comment, link to method, url. Brings up a menu. 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 (resets 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 larger spread)	

1.2. Squeak Smalltalk Syntax: the Basics

1.2.1. 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.

1.2.2. 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).

1.2.3. 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"

1.2.4. 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 undersatnds, in large part, the same operations.
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.

1.2.5. 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.

1.2.6. 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** 1540!) "

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 parsed messages." Binary and unary messages are resolved first unless corrected with parentheses.

1.2.7. 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.

1.2.8. 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!)

1.2.9. 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 MESAGES 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
valueWith Arguments: 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

The message `#value`, sent to a block, causes the block to be executed and answers the result. The block must require zero arguments.

The message `#value: arg`, causes the block to be executed. The block must require exactly one argument; the corresponding parameter is initialized to `arg`.

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`.

1.2.10. 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

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).

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	Same as #ifNil:ifNotNil:	

ifNil: nilBlock		

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

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

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

1.2.11. 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.

1.2.12. Answer Expressions

```
answerExpression ::= ^ expression
^ aTemporary
^ 2+3
```

Inside the body of a method, an answer expression is used to termionate 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.

1.2.13. Class Definition

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'
```

1.2.14. 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
```

2. Squeak Smalltalk: Classes Reference

2.1. Fundamental Classes and Methods

2.1.1. Class Object (Operations on all objects)

INSTANCE CREATION (CLASS SIDE)

Message	Description	Notes
new	Answer a new instance of the receiver (which is a class). This is the usual way of creating a new object. new is often overridden in subclasses to provide class-specific behavior.	1, 2
basicNew	This is the primitive that is ultimately called to implement new.	3
new: anInteger	Answer an instance of the receiver (which is a class) with size given by anInteger. Only allowed if it makes sense to specify a size.	4

Notes:

The usual body for a **new** method is `^ super new initialize`. Remember to put it on the class side, remember to type the ^, and remember to say `super`, not `self`!

It's OK not to implement new if it makes no sense, For example, look at Boolean class>>new and MappedCollection class>>new.

basicNew is there so you can still make instances even if a superclass has overridden **new**. Consequently, never override **basicNew**, until you become a wizard.

If you need an initialization parameter other than a size, choose a more meaningful name than **new:** For example, look at the instance creation protocol for Pen class and Rectangle class.

COMPARING OBJECTS

Message	Description	Notes
== anObject	Are the receiver and the argument the same object? (Answer is true or false)	1
~~ anObject	Are the receiver and the argument different objects?	1
= anObject	Are the receiver and the argument equal? The exact meaning of equality depends on the class of the receiver.	2
~= anObject	Are the receiver and the argument unequal?	2

hash	Answer a SmallInteger whose value is related to the receiver's value.	2

Notes:

== and ~~ should not normally be redefined in any other class.

Since various classes (particularly Sets and Dictionaries) rely on the property that equal objects have equal hashes, you should override `#hash` whenever you override #=. It must be true that `(a = b)` implies `(a hash = b hash)`. The inverse and the converse will not hold in general.

Testing Objects

Message	Description	Notes
isNil	Is the receiver nil? (Answer is true or false)	
notNil	Is the receiver not nil?	
ifNil: aBlock	Evaluate aBlock if the receiver is nil, and answer the value of aBlock. Otherwise answers nil.	
ifNotNil: aBlock	Evaluate aBlock if the receiver is not nil, and answer the value of aBlock. Otherwise answers the receiver.	
ifNotNilDo: aBlock	If the receiver is not nil, evaluate aBlock with the receiver as argument.	1

`ifNotNilDo: aBlock` is useful if the receiver is a complex expression, for example

```
self leftChild rightChild ifNotNilDo: [ :node | node balance ]
```

Copying Objects

Message	Description	Notes
copy	Answer another instance just like the receiver. Subclasses typically override this method; they typically do not override shallowCopy.	
shallowCopy	Answer a copy of the receiver which shares the receiver's instance variables.	
deepCopy	Answer a copy of the receiver with its own copy of each instance variable.	
veryDeepCopy	Do a complete tree copy using a dictionary. An object in the tree twice is copied once and shared by both referents.	

Sending Messages to Objects

Message	Description	Notes
perform: aSymbol	Send the unary selector, aSymbol, to the receiver. Signal an error if the number of arguments expected by the selector is not zero.	
perform:	Send the selector aSymbol to the receiver with anObject	1

aSymbol with: anObject	as its argument. Fail if the number of arguments expected by the selector is not one.	
perform: selector withArguments: argArray	Send the selector, aSymbol, to the receiver with arguments in argArray. Fail if the number of arguments expected by the selector does not match the size of argArray.	

Squeak objects also recognize `#perform:with:with:` and `#perform:with:with:with`

INDEXING OBJECTS

Message	Description	Notes
at: index	Answer the value of an indexable element in the receiver. Signal an Error if index is not an Integer or is out of bounds.	
at: index put: anObject	Store the argument value in the indexable element of the receiver indicated by index. Signal an Error if index is not an Integer or is out of bounds. Or fail if the value is not of the right type for this kind of collection. Answer the value that was stored.	
at: index modify: aBlock	Replace the element at index of the receiver with that element transformed by the block.	
size	Answer the number of indexable variables in the receiver. This value is the same as the largest legal subscript. If the receiver does not have instance variables, then signal an Error.	

DISPLAYING AND STORING OBJECTS

Message	Description	Notes
printString	Answer a String whose characters describe the receiver.	
printOn: aStream	Append to the argument, aStream, a String whose characters describe the receiver.	
storeString	Answer a String from which the receiver can be reconstructed.	
storeOn: aStream	Append to the argument, aStream, a String from which the receiver can be reconstructed	

INTERROGATING OBJECTS

Message	Description	Notes
class	Answers the receiver's class (an object).	
isKindOf: aClass	Is the receiver an instance of aClass or one of its subclasses?	
isMemberO	Is the receiver an instance of aClass? (Same as `rcvr class`	

f: aClass	== aClass)	
respondsTo: aSelector	Can the receiver have find a method for aSelector, either in the receiver's class or in one of its superclasses?	
canUnderstand: aSelector	Does the receiver, which must be a class, have a method for aSelector? The method can belong to the receiver or to any of its superclasses.	

Miscellaneous Messages on Objects

Message	Description	Notes
yourself	Answers self.	1
asString	Answers the receiver's printString.	
doesNotUnderstand: aSymbol	Report that the receiver does not understand aSymbol as a message.	
error: aString	Signal an Error.	
halt	Stops execution.	2

the message `yourself` is mostly used as the last message in a cascade, when the previous message answered some object other than the receiver. For example,

`#(1 2 3 5 5) at: 4 put: 4` answers `4`, the object that was put, whereas

`#(1 2 3 5) at: 4 put: 4; yourself` answers `#(1 2 3 4)`

`self halt` is the usual way of forcing entry to the debugger. The halt can be resumed.

2.1.2. Class Boolean

This abstract class represents logical values, providing Boolean operations and conditional control structures. It has two subclasses, True and False, each of which have singleton instances represented by the Squeak keywords **true** and **false**, respectively.

Evaluating and Non-Evaluating Logical Operations for Boolean

Message	Description	Notes
& aBoolean	Evaluating conjunction (AND). Evaluate the argument. Then answer true if both the receiver and the argument are true.	
eqv: aBoolean	Answer true if the receiver is equivalent to aBoolean.	
not	Negation. Answer true if the receiver is false, answer false if the receiver is true.	
xor: aBoolean	Exclusive OR. Answer true if the receiver is not equivalent to aBoolean.	
| aBoolean	Evaluating disjunction (OR). Evaluate the argument. Then answer true if either the receiver or the argument is true.	

and: alternativeBlock	Nonevaluating conjunction. If the receiver is true, answer the value of the argument, alternativeBlock; otherwise answer false without evaluating the argument.	
or: alternativeBlock	Nonevaluating disjunction. If the receiver is false, answer the value of the argument, alternativeBlock; otherwise answer true without evaluating the argument.	

2.1.3. Class Magnitude

This abstract class embraces, among other classes, Numbers, Characters, Date and Time. It addresses classes whose instances can be linearly ordered.

Message	Description	Notes
< aMagnitude	Answer whether the receiver is strictly less than the argument.	
> aMagnitude	Answer whether the receiver is strictly greater than the argument.	
<= aMagnitude	Answer whether the receiver is less than or equal to the argument.	
>= aMagnitude	Answer whether the receiver is greater than or equal to the argument.	
between: min and: max	Answer whether the receiver is greater than or equal to the argument, min, and less than or equal to the argument, max.	
min: aMagnitude	Answer the receiver or the argument, whichever is the lesser magnitude.	
max: aMagnitude	Answer the receiver or the argument, whichever is the greater magnitude.	
min: firstMagnitude max: secondMagnitude	Take the receiver or the argument, firstMagnitude, whichever is the lesser magnitude, and answer that or the argument, secondMagnitude, whichever is the greater magnitude.	

2.1.4. Class Character

Instances of class character store the associated 8-bit extended ASCII code. Character has the unique instance property, so that all equal ("=") instances of a character are identical ("=="). That is, `a asciiValue == b asciiValue` if and only if `a == b`.

METHODS FOR ACCESSING CHARACTERS

Message	Description	Notes

asciiValue	Answer the value of the receiver that represents its ASCII encoding.	
digitValue	Answer 0-9 if the receiver is $0-$9, 10-35 if it is $A-$Z, and < 0 otherwise. This is used to parse literal numbers of radix 2-36.	

METHODS FOR TESTING CHARACTERS

Message	Description	Notes
isAlphaNumeric	Answer whether the receiver is a letter or a digit.	
isDigit	Answer whether the receiver is a digit.	
isLetter	Answer whether the receiver is a letter.	
isLowercase	Answer whether the receiver is a lowercase letter.	
isSeparator	Answer whether the receiver is one of the separator characters: space, cr, tab, line feed, or form feed.	
isSpecial	Answer whether the receiver is one of the special characters	
isUppercase	Answer whether the receiver is an uppercase letter.	
isVowel	Answer whether the receiver is one of the vowels, AEIOU, in upper or lower case.	
tokenish	Answer whether the receiver is a valid token-character--letter, digit, or colon.	

METHODS FOR CONVERTING CHARACTERS

Message	Description	Notes
asLowercase	If the receiver is uppercase, answer its matching lowercase Character.	
asUppercase	If the receiver is lowercase, answer its matching uppercase Character.	

2.2. Numeric Classes and Methods

2.2.1. Class Number

This abstract class embraces Integers, Floats and Fractions. Number is a subclass of Magnitude.

METHODS FOR ARITHMETIC ON ALL NUMERIC CLASSES

Message	Description	Notes
+ aNumber	Answer the sum of the receiver and the argument.	

- aNumber	Answer the difference of the receiver and the argument.	
* aNumber	Answer the product of the receiver and the argument.	
/ aNumber	Answer the result of dividing the receiver and the argument, retaining as much precision as possible. If the answer is not exact, the result will be Fraction or Float, as appropriate. Signal ZeroDivide if the argument is Zero.	
// aNumber	Answer the result of dividing the receiver and the argument, truncating toward negative infinity. Signal ZeroDivide if the argument is Zero.	
\\ aNumber	Answer the remainder of dividing the receiver and the argument, truncating toward negative infinity. This is the modulus operator. Signal ZeroDivide if the argument is Zero.	
quo: aNumber	Answer the result of dividing the receiver and the argument, truncating toward zero. Signal ZeroDivide if the argument is Zero.	
rem: aNumber	Answer the remainder of dividing the receiver and the argument, truncating toward zero. Signal ZeroDivide if the argument is Zero.	
abs	Answer the absolute value of the receiver.	
negated	Answer the negation of the receiver.	
reciprocal	Answer 1 divided by the receiver. Signal ZeroDivide if the receiver is zero.	

METHODS IMPLEMENTING MATHEMATICAL FUNCTIONS FOR NUMBERS

Message	Description	Notes
exp	Answer a floating point number that is the exponential of the reciever	
ln	Answer the natural log of the receiver.	
log: aNumber	Answer the logarithm base aNumber of the reciever.	
floorLog: aNumber	Take the logarithm base aNumber of the reciever, and answer the integer nearest that value towards negative infinity.	
raisedTo: aNumber	Answer the receiver raised to the power of the argument, aNumber.	
raisedToInteger: anInteger	Answer the receiver raised to the power of the argument, anInteger. Signal an Error if anInteger is not an integer.	
sqrt	Answer a floating point number that is the positive	

	square root of the receiver.	
squared	Answer the receiver multiplied by itself.	

Methods for testing Numbers

Message	Description	Notes
even	Answer whether the receiver is even.	
odd	Answer whether the receiver is odd.	
negative	Answer whether the receiver is less than zero.	
positive	Answer whether the receiver is greater than or equal to zero.	
strictlyPositive	Answer whether the receiver is greater than zero.	
sign	Answer 1 if the receiver is strictly positive, zero if the receiver is zero, and -1 if the receiver is strictly negative.	
isZero	Answer whether the receiver is zero.	

Methods for truncating and rounding Numbers

Message	Description	Notes
ceiling	Answer the Integer nearest the receiver toward positive infinity.	
floor	Answer the Integer nearest the receiver toward negative infinity.	
truncated	Answer an integer nearest the receiver toward zero.	
truncateTo: aNumber	Answer the next multiple of aNumber toward zero that is nearest the receiver.	
rounded	Answer the integer nearest the receiver.	
roundTo: quantum	Answer the nearest number that is a multiple of quantum.	
roundUpTo: quantum	Answer the next multiple of aNumber toward infinity that is nearest the receiver.	

Methods for trigonometry on Numbers

Message	Description	Notes
sin	Answer the sine of the receiver taken as an angle in radians.	
cos	Answer the cosine of the receiver taken as an angle in radians.	
tan	Answer the tangent of the receiver taken as an angle in radians.	
degreeSin	Answer the sin of the receiver taken as an angle in degrees.	

degreeCos	Answer the cosine of the receiver taken as an angle in degrees.	
arcSin	Answer an angle in radians whose sine is the receiver.	
arcCos	Answer an angle in radians whose cosine is the receiver.	
arcTan	Answer an angle in radians whose tangent is the receiver.	
arcTan: denominator	Answer the angle in radians whose tan is the receiver divided by denominator.	
degreesToRadians	Answer the receiver in radians. Assumes the receiver is in degrees.	
radiansToDegrees	Answer the receiver in degrees. Assumes the receiver is in radians.	

2.3. Collection Classes and Methods

2.3.1. The Collection Hierarchy

Class	Description
Collection	Abstract Class for Collections
Bag	Unordered, unindexed collection of objects
Set	Unordered, unindexed collection of unique objects
Dictionary	Set of associations (values are indexable by keys)
IdentityDictionary	Dictionary, but comparisons are done using ==
IdentitySet	Set, but comparisons are done using ==
SequenceableCollection	Ordered collection, indexed by integers
OrderedCollection	Ordered according to manner elements are added and removed
SortedCollection	Ordered according to value of a "sortBlock"
LinkedList	Homogeneous SequenceableCollection of Links
Interval	Homogeneous sequence of arithmetic progression of Integers
ArrayedCollection	Ordered collection, indexed by fixed range of Integers
Array	ArrayedCollection of arbitrary Objects
Array2D	Homogeneous ArrayedCollection of Arrays

ByteArray	Homogeneous ArrayedCollection of Bytes (Integers -128..255)
FloatArray	Homogeneous ArrayedCollection of Floating point numbers
IntegerArray	Homogeneous ArrayedCollection of Signed 32-bit Integers
PointArray	Homogeneous ArrayedCollection of Points (with 32-bit values)
RunArray	Homogeneous ArrayedCollection of Integers (sparse RLE representation)
ShortIntegerArray	Homogeneous ArrayedCollection of Signed 16-bit Integers
ShortPointArray	Homogeneous ArrayedCollection of Points (with 16-bit values)
ShortRunArray	Homogeneous ArrayedCollection of Signed 16-bit Ints (sparse RLE rep)
String	Homogeneous ArrayedCollection of Characters
Symbol	Homogeneous ArrayedCollection of Characters (with unique instance property)
Text	Homogeneous ArrayedCollection of Characters with associated text attributes
WordArray	Homogeneous ArrayedCollection of Unsigned 32-bit Integers
Heap	Like SortedCollection, but stores information as a heap. (see Heapsort)
MappedCollection	Means for accessing an indexable Collection, using a mapping from a collection of "external" keys to the accessed collection's "indexing" keys. The MappedCollection can then be used directly, indexing and changing the accessed collection via the external keys.

2.3.2. Class Collection

Methods for creating Collections (Class Side)

Message	Description	Notes
with: anObject	Answer an instance of the receiver containing anObject	
with: firstObject	Answer an instance of the receiver containing all the arguments as elements. (Squeak recognizes instantiators of	

with: secondObject	this type up to six "with:" clauses).	
withAll: aCollection	Answer an instance of the receiver containing all the elements from aCollection.	

METHODS FOR TESTING, ADDING AND REMOVING COLLECTION ELEMENTS

Message	Description	Notes
anyOne	Answer a specimen element of the receiver (any one at all). Signal an error if the receiver is empty.	
isEmpty	Answer whether the receiver contains any elements.	
occurrencesOf: anObject	Answer how many of the receiver's elements are equal to anObject.	
anySatisfy: aBlock	Evaluate aBlock with the elements of the receiver. If aBlock returns true for any element return true. Otherwise return false	
includes: anObject	Answer whether anObject is one of the receiver's elements.	
includesAllOf: aCollection	Answer whether all the elements of aCollection are in the receiver.	
includesAnyOf: aCollection	Answer whether any element of aCollection is one of the receiver's elements.	
difference: secondCollection	Answer a new collection that is computed by copying the receiver and removing all the elements in secondCollection.	
add: newObject	Include newObject as one of the receiver's elements. Answer newObject. ArrayedCollections cannot respond to this message.	
addAll: newObject	Include all the elements of aCollection as the receiver's elements. Answer aCollection.	
remove: oldObject	Remove oldObject as one of the receiver's elements. Answer oldObject unless no element is equal to oldObject, in which case, signal an Error.	
remove: oldObject ifAbsent: anExceptionBlock	Remove oldObject as one of the receiver's elements. If several of the elements are equal to oldObject, only one is removed. If no element is equal to oldObject, answer the result of evaluating anExceptionBlock. Otherwise, answer oldObject. SequenceableCollections cannot respond to this message.	

removeAll: aCollection	Remove each element of aCollection from the receiver. If successful for each, answer aCollection. Otherwise signal an Error.	
removeAllFoundIn: aCollection	Remove from the receiver each element of aCollection that is present in the receiver.	
removeAllSuchThat: aBlock	Apply the condition to each element and remove it if the condition is true.	

METHODS FOR ENUMERATING COLLECTIONS

Message	Description	Notes
do: aBlock	Evaluate aBlock with each of the receiver's elements as the argument.	
do: aBlock separatedBy: separatorBlock	Evaluate aBlock for all elements in the receiver, and if there is more than one element, evaluate the separatorBlock between each pair of elements in the receiver.	
select: aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver, only those elements for which aPredicateBlock evaluates to true. Answer the new collection.	
reject: aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver only those elements for which aPredicateBlock evaluates to false. Answer the new collection.	
collect: aMappingBlock	Evaluate aMappingBlock with each of the receiver's elements as the argument. Collect the resulting values into a collection like the receiver. Answer the new collection.	
detect: aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Answer the first element for which aPredicateBlock answers true. Signal an Error if none are found.	
detect: aPredicateBlock ifNone: exceptionBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Answer the first element for which aPredicateBlock evaluates to true. If there is none, answer the result of evaluating exceptionBlock.	
inject: initialValue into:	Accumulate a running value associated with evaluating binaryBlock. The running value is initialized to	

binaryBlock	initialValue. The current running value and the next element of the receiver are provided as the arguments to binaryBlock. For example, to compute the sum of the elements of a numeric collection, aCollection inject: 0 into: [:subTotal :next | subTotal + next].	
collect: aMappingBlock thenSelect: aPredicateBlock	Evaluate aMappingBlock with each of the receiver's elements as the argument. Collect the resulting values that satisfy aPredicateBlock into a collection like the receiver. Answer the new collection.	
select: aPredicateBlock thenCollect: aMappingBlock	Evaluate aMappingBlock with each of the receiver's elements for which aPredicateBlock answers true as the argument. Collect the resulting values into a collection like the receiver. Answer the new collection.	
count: aPredicateBlock	Evaluate aPredicateBlock with each of the receiver's elements as the argument. Return the number that answered true.	

2.3.3. Bag

Methods for accessing Bags

Message	Description	Notes
add: newObject withOccurrences: anInteger	Add the element newObject to the receiver. Do so as though the element were added anInteger number of times. Answer newObject.	

2.3.4. Dictionary and IdentityDictionary

Methods for Accessing Dictionaries

Dictionaries are homogenous Sets of key and value pairs. These pairs are called Associations: key and value can be any object. Instances of Association are created by sending the binary message "`key -> value`" (`->` is defined in Object). Dictionaries have the property that each key occurs at most once. IdentityDictionaries have the same property, but determine uniqueness of keys using == instead of =. In ordinary use, both kinds of Dictionary are indexed using the unique key to obtain the corresponding value.

Message	Description	Notes
at: aKey	Answer the value associated with aKey. Signal an Error if no value is associated with aKey.	
at: aKey ifAbsent: aBlock	Answer the value associated with aKey. If no value is associated with aKey, answer the value of aBlock.	
associationAt:	Answer the association whose key is aKey. If there is	

aKey	none, signal an Error	
associationAt: aKey ifAbsent: aBlock	Answer the association whose key is aKey. If there is none, answer the value of aBlock.	
keyAtValue: aValue	Answer the key of the first association having aValue as its value. If there is none, signal an Error.	
keyAtValue: aValue ifAbsent: exceptionBlock	Answer the key of the first associaiton having aValue as its value. If there is none, answer the result of evaluating exceptionBlock.	
keys	Answer a Set containing the receiver's keys.	
values	Answer an Array containing the receiver's values.	
includes: aValue	Does the receiver contain a value equal to aValue?	
includesKey: aKey>	Does the receiver have a key equal to aKey?	
do: aBlock	Evaluate aBlock with each of the receiver's values as argument.	
keysDo: aBlock	Evaluate aBlock with each of the receiver's keys as argument.	
valuesDo: aBlock	same as do:	
keysAndValuesDo: aBinaryBlock	Evaluate aBinaryBlock with each of the receiver's keys and the associated value as the two arguments.	
associationsDo: aBlock	Evaluate aBlock with each of the receiver's elements (key/value associations) as the argument.	

2.3.5. Sequenceable Collection

METHODS FOR ACCESSING SEQUENCEABLECOLLECTIONS

Message	Description	Notes
atAll: indexCollection	Answer a collection containing the elements of the receiver specified by the integer elements of the argument, indexCollection.	
atAll: aCollection put: anObject	Put anObject at every index specified by the integer elements of the argument, aCollection.	
atAllPut: anObject	Put anObject at every one of the receiver's indices.	
first	Answer the first element of the receiver. (Squeak also recognizes second, third, fourth, fifth and sixth). Signal an error if there aren't sufficient elements in the receiver.	
middle	Answer the median element of the receiver.	

	Signal an error if the receiver is empty.	
last	Answer the last element of the receiver. Signal an error if the receiver is empty.	
allButFirst	Answer a collection equal to the reciever, but without the first element. Signal an error if the receiver is empty.	
allButLast	Answer a collection equal to the reciever, but without the last element. Signal an error if the receiver is empty.	
indexOf: anElement	Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer 0.	
indexOf: anElement ifAbsent: exceptionBlock	Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer the result of evaluating the argument, exceptionBlock.	
indexOfSubCollection: aSubCollection startingAt: anIndex	Answer the index of the receiver's first element, such that that element equals the first element of aSubCollection, and the next elements equal the rest of the elements of aSubCollection. Begin the search at element anIndex of the receiver. If no such match is found, answer 0.	
indexOfSubCollection: aSubCollection startingAt: anIndex ifAbsent: exceptionBlock	Answer the index of the receiver's first element, such that that element equals the first element of sub, and the next elements equal the rest of the elements of sub. Begin the search at element start of the receiver. If no such match is found, answer the result of evaluating argument, exceptionBlock.	
replaceFrom: start to: stop with: replacementCollection	This destructively replaces elements from start to stop in the receiver. Answer the receiver itself. Use copyReplaceFrom:to:with: for insertion/deletion that may alter the size of the result.	
replaceFrom: start to: stop: with: replacementCollection startingAt: repStart	This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the sequenceable collection, replacementCollection. Answer the receiver. No range checks are performed.	

METHODS FOR COPYING SEQUENCEABLECOLLECTIONS

Message	Description	Notes
, otherCollection	Answer a new collection comprising the receiver concatenated with the argument, otherCollection.	
copyFrom: start to: stop	Answer a copy of a subset of the receiver that contains all the elements between index start and index stop, inclusive of both.	
copyReplaceAll: oldSubCollection with: newSubCollection	Answer a copy of the receiver in which all occurrences of oldSubstring have been replaced by newSubstring.	
copyReplaceFrom: start to: stop with: replacementCollection	Answer a copy of the receiver satisfying the following conditions: If stop is less than start, then this is an insertion; stop should be exactly start-1, start = 1 means insert before the first character, start = size+1 means append after last character. Otherwise, this is a replacement; start and stop have to be within the receiver's bounds.	
copyWith: newElement	Answer a copy of the receiver that is 1 bigger than the receiver and has newElement at the last element.	
copyWithout: oldElement	Answer a copy of the receiver in which all occurrences of oldElement have been left out.	
copyWithoutAll: aList	Answer a copy of the receiver in which all occurrences of all elements in aList have been removed.	
forceTo: length paddingWith: anElement	Force the length of the collection to length, padding if necissary with elem. Note that this makes a copy.	
reversed	Answer a copy of the receiver in which the sequencing of all the elements has been reversed.	
shuffled	Answer a copy of the receiver in which the elements have been permuted randomly.	
sortBy: aBlock	Create a copy that is sorted. Sort criteria is the block that accepts two arguments. When the block is true, the first arg goes first ([:a :b | a > b] sorts in descending order).	

METHODS FOR ENUMERATING SEQUENCEABLECOLLECTIONS

Message	Description	Notes
findFirst: aBlock	Return the index of the receiver's first element for	

	which aBlock evaluates as true.	
findLast: aBlock	Return the index of the receiver's last element for which aBlock evaluates as true.	
keysAndValuesDo: aBinaryBlock	Evaluate aBinaryBlock once with each valid index for the receiver in order, along with the corresponding value in the receiver for that index.	
reverseDo: aBlock	Evaluate aBlock with each of the receiver's elements as the argument, starting with the last element and taking each in sequence up to the first. For SequenceableCollections, this is the reverse of the enumeration for #do:.	
with: otherCollection do: binaryBlock	Evaluate binaryBlock with corresponding elements from this collection and otherCollection.	
reverseWith: otherCollection do: aBinaryBlock	Evaluate aBinaryBlock with each of the receiver's elements, in reverse order, along with the corresponding element, also in reverse order, from otherCollection.	

2.3.6. OrderedCollections

METHODS FOR ACCESING ORDEREDCOLLECTIONS

Message	Description	Notes
add: newObject before: oldObject	Add the argument, newObject, as an element of the receiver. Put it in the sequence just preceding oldObject. Answer newObject.	
add: newObject after: oldObject	Add the argument, newObject, as an element of the receiver. Put it in the sequence just succeeding oldObject. Answer newObject.	
add: newObject afterIndex: index	Add the argument, newObject, as an element of the receiver. Put it in the sequence just after index. Answer newObject.	
addFirst: anElement	Add newObject to the beginning of the receiver. Answer newObject.	
addAllFirst: anOrderedCollection	Add each element of anOrderedCollection at the beginning of the receiver. Answer anOrderedCollection.	
addLast: anElement	Add newObject to the end of the receiver. Answer newObject.	
addAllLast: anOrderedCollection	Add each element of anOrderedCollection at the end of the receiver. Answer anOrderedCollection.	

removeAt: anIndex	remove the element of the receiver at location anIndex. Answers the element removed.	
removeFirst	Remove the first element of the receiver and answer it. If the receiver is empty, signal an Error.	
removeLast	Remove the last element of the receiver and answer it. If the receiver is empty, signal an Error.	

2.3.7. Strings

String is an extensive class, built over the ages in something of an ad hoc manner. We describe here only a small fraction of the methods provided (there are about 300!)

METHODS FOR ACCESSING STRINGS

Message	Description	Notes
findAnySubStr: delimiters startingAt: start	Answer the index of the character within the receiver, starting at start, that begins a substring matching one of the delimiters; delimiters is an Array of Strings and/or Characters. If the receiver does not contain any of the delimiters, answer size + 1.	
findBetweenSubStrs: delimiters	Answer the collection of tokens that results from parsing the receiver. And of the Strings (or Characters) in the Array delimiters is recognized as separating tokens.	
findDelimiters: delimiters startingAt: start	Answer the index of the character within the receiver, starting at start, that matches one of the delimiters. If the receiver does not contain any of the delimiters, answer size + 1.	
findString: subString	Answer the first index of subString within the receiver. If the receiver does not contain subString, answer 0.	
findString: subString startingAt: start	Answer the index of subString within the receiver, starting at start. If the receiver does not contain subString, answer 0.	
findTokens: delimiters	Answer the collection of tokens that results from parsing the receiver. Any character in the argument, delimiters, marks a border. Several delimiters in a row are considered as just one separator	
indexOf: aCharacter	Answer the index of the first occurrence of aCharacter in the receiver. 0 Otherwise.	
indexOf: aCharacter	Answer the index of the first occurrence of	

startingAt: start	aCharacter in the receiver, beginning at index start. 0 Otherwise.	
indexOf: aCharacter startingAt: start ifAbsent: aBlock	Answer the index of the first occurrence of aCharacter in the receiver, beginning at index start. If not present, answer the value of aBlock.	
indexOfAnyOf: aCharacterSet	Answers the index of the first occurrence in the receiver of any character in the given set. Returns 0 if none is found.	1

Notes

As with #indexOf:, there are corresponding messages #indexOfAnyOf:ifAbsent:, #indexOfAnyOf:startingAt: and #indexOfAnyOf:startingAt:ifAbsent:)

METHODS FOR COMPARING STRINGS

Message	Description	Notes
= aString	Answer whether the receiver is equal to aString.The comparison is case-sensitive,	
< aString, <= aString > aString >= aString	Answer whether the receiver sorts as indicated with aString. The collation order is that of the Squeak character set, and therefore case-sensitive,	
sameAs: aString	Answer whether the receiver is equal to aString, ignoring differences of case.	
compare: aString	Answer a code defining how the receiver sorts relative to the argument, aString. 1 - receiver before aString; 2 - receiver equal to aString; and 3 - receiver after aString. The collation sequence is that of the Squeak character set and is case insensitve.	
match: text	Answer whether text matches the pattern in the receiver. Matching ignores upper/lower case differences. Where the receiver contains #, text may contain any character. Where the receiver contains *, text may contain any sequence of characters.	
beginsWith: prefix	Answer whether the receiver begins with the argument, prefix.	
endsWith: prefix	Answer whether the receiver ends with the argument, prefix.	
alike: aString	Answer a non-negative integer indicating how similar the receiver is to aString. 0 means "not at all alike". The best sccore is `aString size * 2`.	

METHODS FOR CONVERTING STRINGS

Message	Description	Notes
asLowercase	Answer a new String that matches the receiver but without any upper case characters.	
asUppercase	Answer a new String that matches the receiver but without any lower case characters.	
capitalized	Answer a copy of the receiver with the first character capitalized if it is a letter.	
asDisplayText	Answer a copy of the receiver with default font and style information.	
asInteger	Attempts to parse the receiver as an Integer. Answers the Integer, or nil if the receiver does not start with a digit.	
asNumber	Attempts to parse the receiver as a Number. It is an error if the receiver does not start with a digit.	
asDate	Attempts to parse the receiver as adate, and answers an appropriate instance of class Date. Many formats are recognized.	

2.3.8. Class Integer

METHODS FOR ARITHMETIC ON INTEGERS

Message	Description	Notes
isPowerOfTwo	Answer whether the receiver is a power of two.	
factorial	Answer the factorial of the receiver.	
gcd: anInteger	Answer the greatest common denominator of the receiver and the argument.	
lcm: anInteger	Answer the least common multiple of the receiver and the argument.	
take: anInteger	Answer the number of combinations of the receiver, taken the argument, anInteger, at a time.	

METHODS FOR BIT MANIPULATION ON INTEGERS

A range of bit manipulation operations are available on Integers. They are rarely needed, however, so they are not described here. Of course, they can be viewed using the browser.

2.4. Streaming Classes and Methods

2.4.1. The Stream Hierarchy

Class	Description
Stream	Abstract Class for Accessors
PositionableStream	Accessors for Collections Indexable by an Integer
ReadStream	Read-Only
WriteStream	Write-Only
ReadWriteStream	Read and/or Write
FileStream	Accessors for collections whose elements are "paged in"
StandardFileStream	Accessors for files accessed from a file system
CrLfFileStream	Automatically handles system-specific line endings
DummyStream	Like /dev/null

2.4.2. Class Stream

Stream is an abstract class for an accessor to a sequence of objects, called the contents. The stream is said to be "advanced" when the stream is configured to access a later element of the contents.

METHODS FOR ACCESSING STREAMS

Message	Description	Notes
contents	Answer the entire contents of the receiver.	
next	Answer the next object accessible by the receiver.	
next: anInteger	Answer the next anInteger number of objects accessible by the receiver.	
next: n put: anObject	Make the next n objects accessible by the receiver anObject. Answer anObject.	
nextMatchAll: aColl	Answer true if next N objects are the ones in aColl, else false. Advance stream if true, leave as was if false.	
nextMatchFor: anObject	Answer whether the next object is equal to the argument, anObject, advancing the stream.	
nextPut: anObject	Insert the argument, anObject, as the next object accessible by the receiver. Answer anObject.	
nextPutAll: aCollection	Append the elements of aCollection to the sequence of objects accessible by the receiver. Answer aCollection.	

upToEnd	Answer the remaining elements in the string	
flush	Ensure that any objects buffered in the receiver are sent to their final destination.	

METHODS FOR TESTING STREAMS

Message	Description	Notes
atEnd	Answer whether the receiver can access any more objects.	

METHODS FOR ENUMERATING STREAMS

Message	Description	Notes
do: aBlock	Evaluate aBlock for each of the remaining objects accessible by receiver.	

2.4.3. Class PositionableStream

PositionableStream is an abstract class for accessors to sequences of objects that can be externally named by indices so that the point of access can be repositioned. Concrete classes ReadStream, WriteStream and ReadWriteStream are typically used to instantiate a PositionableStream on Collections, depending upon the access mode. StandardFileStream and CRLFFileStream are typically used for instantiating PositionableStreams for Files.

METHODS FOR ACCESSING POSITIONABLESTREAMS

Message	Description	Notes
contentsOfEntireFile	Answer a collection containing the remainder of the receiver.	
last	Return the final element of the receiver.	
nextDelimited: terminator	Answer the contents of the receiver, from the current position up to the next terminator character; provided, however, that doubled terminators will be included as a single element.	
nextInto: buffer	Given buffer, an indexable object of size n, fill buffer with the next n objects of the receiver.	
nextLine	Answer next line (may be empty), or nil if at end	
originalContents	Answer the receiver's actual contents collection. (contents returns a copy)	
peek	Answer what would be returned if the message next were sent to the receiver, but don't advance the receiver. If the receiver is at the end, answer nil.	
peekFor: anObject	Answer false and do not move over the next	

	element if it is not equal to anObject, or if the receiver is at the end. Answer true and advance the stream if the next element is equal to anObject.	
upTo: anObject	Answer a subcollection from the current access position to the occurrence (if any, but not inclusive) of anObject in the receiver. If anObject is not in the collection, answer the entire rest of the receiver.	
upToAll: aCollection	Answer a subcollection from the current access position to the occurrence (if any, but not inclusive) of aCollection. If aCollection is not in the stream, answer the entire rest of the stream.	

METHODS FOR TESTING POSITIONABLESTREAMS

Message	Description	Notes
isEmpty	Answer whether the receiver's contents has no elements.	

METHODS FOR POSITIONING POSITIONABLESTREAMS

Message	Description	Notes
match: subCollection	Set the access position of the receiver to be past the next occurrence of the subCollection. Answer whether subCollection is found. No wildcards, case sensitive.	
padTo: nBytes put: aCharacter	Pad, using aCharacter, to the next boundary of nBytes.	
padToNextLongPut: char	Make position be on long word boundary, writing the padding character, char, if necessary.	
position	Answer the current position of accessing the sequence of objects.	
position: anInteger	Set the current position for accessing the objects to be anInteger, as long as anInteger is within the bounds of the receiver's contents. If it is not, create an error notification.	
reset	Set the receiver's position to the beginning of the sequence of objects.	
resetContents	Set the position and limits to 0.	
setToEnd	Set the position of the receiver to the end of the sequence of objects.	
skip: anInteger	Set the receiver's position to be the current	

	position+anInteger. A subclass might choose to be more helpful and select the minimum of the receiver's size and position+anInteger, or the maximum of 1 and position+anInteger for the repositioning.	
skipTo: anObject	Set the access position of the receiver to be past the next occurrence of anObject. Answer whether anObject is found.	

2.4.4. Class WriteStream

METHODS FOR WRITING CHARACTERS ON WRITESTREAMS

Message	Description	Notes
cr	Append a return character to the receiver.	
crtab	Append a return character, followed by a single tab character, to the receiver.	
crtab: anInteger	Append a return character, followed by anInteger tab characters, to the receiver.	
space	Append a space character to the receiver.	
tab	Append a tab character to the receiver.	

2.5. ANSI-Compatible Exceptions

2.5.1. Evaluating Blocks with Exceptions

METHODS FOR HANDLING EXCEPTIONS RAISED IN A BLOCKCONTEXT

Message	Description	Notes
ensure: aTerminationBlock	Evaluate aTerminationBlock after evaluating the receiver, regardless of whether the receiver's evaluation completes.	
ifCurtailed: aTerminationBlock	Evaluate the receiver. If it terminates abnormally, evaluate aTerminationBlock.	
on: exception do: handlerActionBlock	Evaluate the receiver in the scope of an exception handler, handlerActionBlock.	

EXAMPLES

```
 ["target code, which may abort"]     ensure:         ["code that will
always be executed              after the target code,
whatever whatever may happen"]          ["target code, which may abort"]
ifCurtailed:         ["code that will be executed             whenever the
```

```
target code terminates                  without a normal return"]
["target code, which may abort"]     on: Exception     do: [:exception |
"code that will be executed whenever                the identified Exception
is signaled."]
```

2.5.2. Exceptions

Exception is an abstract class; instances should neither be created nor trapped. There are two common subclasses of Exception, Error and Notification, from which subclasses normally inherit. Errors are not resumable; a Notification is an indication that something interesting has occurred; if it is not handled, it will pass by without effect.
Exceptions play two distinct roles: that of the exception, and that of the exception handler.

METHODS FOR DESCRIBING EXCEPTIONS

Message	Description	Notes
defaultAction	The default action taken if the exception is signaled.	
description	Return a textual description of the exception.	
isResumable	Determine whether an exception is resumable.	
messageText	Return an exception's message text.	
tag	Return an exception's tag value.	

METHODS FOR SIGNALLING EXCEPTIONS

Message	Description	Notes
signal	Signal the occurrence of an exceptional condition.	
signal: signalerText	Signal the occurrence of an exceptional condition with a specified textual description.	

METHODS FOR DEALING WITH A SIGNALED EXCEPTION

Message	Description	Notes
isNested	Determine whether the current exception handler is within the scope of another handler for the same exception.	
outer	Evaluate the enclosing exception action for the receiver and return.	
pass	Yield control to the enclosing exception action for the receiver.	
resignalAs: replacementException	Signal an alternative exception in place of the receiver.	
resume	Return from the message that signaled the receiver.	
resume: resumptionValue	Return the argument as the value of the message that signaled the receiver.	
retry	Abort an exception handler and re-evaluate its	

	protected block.	
retryUsing: alternativeBlock	Abort an exception handler and evaluate a new block in place of the handler's protected block.	
return	Return nil as the value of the block protected by the active exception handler.	
return: returnValue	Return the argument as the value of the block protected by the active exception handler.	

2.5.3. Class ExceptionSet

An ExceptionSet is used to specify a set of exceptions for an exception handler.

CREATING EXCEPTIONSET

Message	Description	Notes
, anException	Receiver may be an Exception class or an ExceptionSet. Answers an exception set that contains the receiver and anException.	

EXAMPLE

```
["target code, which may abort"]     on: Exception, Error, ZeroDivide
do:         [:exception |             "code that will be executed whenever
one of the identified Exceptions is             signaled."]
```

2.6. The Squeak Class Hierarchy

In Smalltalk, "everything is an object." That is, everything is an instance of class Object or an instance of some subclass of class Object. Everything. Numbers, Classes, Metaclasses, everything. I refer to this as the "Object rule."
Actually, Squeak bends this rule a little bit; the Object rule does not apply for certain system objects, which derive from class ProtoObject. Nevertheless, except for these few system objects, the vast majority of Squeak objects, which I call, "proper objects," satisfy the Object Rule. Proper Objects and their classes and metaclasses, satisfy the following properties.

2.6.1. The Laws of Proper (Smalltalk) Classes

Every proper class is a subclass of class Object, except for Object itself, which has no proper superclass. In particular, Class is a subclass of ClassDescription, which is a subclass of Behavior which is a subclass of Object.
Every object is an instance of a class.
Every class is an instance of a metaclass.
All metaclasses are (ultimately) subclasses of Class.
Every metaclass is an instance of MetaClass.

The methods of Class and its superclasses support the behavior common to those objects that are classes.
The methods of instances of MetaClass add the behavior specific to particular classes.

2.6.2. Class ProtoObject

Squeak additionally supports an improper class ProtoObject, from which object hierarchies other than proper instances and proper classes can inherit. ProtoObject is the superclass of class Object and has no instances. Presently, there are two subclasses of ProtoObject besides Object: ObjectOut and ImageSegmentRootStub, both of which are used to do magic involving objects that have been moved out of memory onto an external medium. You might need to subclass ProtoObject if you are doing something like implementing a remote message send system where you have proxies for remote objects (those on another computer). However, as with proper classes, ProtoObject, is an instance of a metaclass, ProtoObject class, which in turn is an instance of class MetaClass.

2.7. Categories of Squeak Classes

This quick reference only scratches the surface of the functionality available through Squeak. To assist the beginner in surveying the system, the following outline, or roadmap, of the Squeak system is provided.

Category	Description
Kernel	Primary Smalltalk classes for creating and manipulating Smalltalk objects, the Object hierarchy, coroutines and parallel processes. Subcategories include: Objects, Classes, Methods and Processes.
Numeric	Classes for numeric operations, including date and time operations. Subcategories include Magnitudes and Numbers
Collections	Classes for aggregations of Smalltalk objects.
Graphics	Core classes for Smalltalk graphic objects as well as facilities and applications for operating on graphic objects. Key classes include Form and BitBlt.
Interface	The "traditional" MVC User Interface Framework. Also found here are a number of useful Smalltalk applications, including: Squeak browsers, a mail client, a web browser, irc chat client and facilities for operating on "projects."
Morphic	Squeak's Morphic User Interface Framework
Music	Classes supporting Squeak's Sound Synthesis capabilities. Also found here are several useful facilities and applications for manipulating MIDI data and other representations of musical scores.

System	Key System Facilities. Subclasses include: Compiler (Smalltalk compiler); Object Storage (virtual memory for Smalltalk objects); File facilities; Compression; Serial Data Transmission; Basic network facilities.
Exceptions	Class supporting Squeak's ANSI-compliant exceptions facilities.
Network	Classes implementing various Internet and Squeak related network protocols.
PluggableWebServer	A complete web-server application, including an implementation of SWIKI, a collaborative world-wide-web environment. Key classes include: PWS
HTML	Classes for manipulating HTML data.
Squeak	Here lives the mouse. Key classes include: the Squeak VM and an interpreter; the Squeak Smalltalk Subset (Slang) to C translator; and facilities for developing native plugins (pluggable primitives).
Balloon	Classes for complex 2-D graphic objects and fast 2-D graphics rendering.
Balloon-3D	Classes for complex 3-D graphics objects and fast 3-D graphics rendering.
TrueType	Classes for manipulating and displaying TrueType data.
MM-Flash	Classes for manipulating and displaying Flash file data.
Alice & Wonderland	A remarkable interactive 3-D graphics environment.