GeneralClasses

COLLABORATORS

	TITLE : GeneralClasses	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 10, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

GeneralClasses

1.1 Descriptions of the Methods of the General classes:

WARNING: The documentation in this file is from the Original Little
SmallTalk documentation. If there is any question of whether
these documents are correct, you should check the corresponding
source file in AmigaTalk:General/ directory in order
to determine what is currently implemented.
Show below is the hierarchy of the General Classes that are loaded
into memory before the AmigaTalk system is ready for user input.
The indentations indicate which classes are sub-classes:

Object
UndefinedObject
Symbol
Boolean
True
False
Magnitude
Char
Number
Integer
Float
Radian
Point
Random
Collection
Bag
Set

KeyedCollection

Dictionary

AmigaTalk

File

SequenceableCollection

Interval

LinkedList

Semaphore

Form -- Do NOT use!

Pen

ArrayedCollection

Array

ByteArray

String

Block

Class

Process

1.2 Pen Class:

The class Pen is a class that opens a Window for performing simple graphics commands in. This class has been re-written & is completely different from the intentions of the Little SmallTalk author, Tim Budd. Instead of using a plotting device (How many of those are there for the Amiga?), this class simply opens a Window that can be used to see the results of the Pen methods.

NOTE: There's a limit of 20 for how many Plot Windows can be open at the same time. AmigaTalk will tell you via Requesters when this limit is violated.

Responds to

new

make a new instance of class Pen, initializing the instance variables (default title: 'Unknown Plot').

new: newPlotTitle

make a new instance of class Pen, initializing the instance variables & using the supplied newPlotTitle as the Plot Window title.

openPlotEnv: sizePoint

Open the Plot Window with the given size (sizePoint is of class Point ,

so (sizePoint x) is the width, & (sizePoint y) is the height of the
Plot Window).
WARNING: You can only open a Plot Window as big as the AmigaTalk screen
(default 640 by 480).
closePlotEnv: whichPlotTitle
Close the Plot Window with the given title.
movePlotEnvBy: deltaPoint
Move the Plot Window by the given deltaPoint amounts (deltaPoint is
of class Point , so (deltaPoint x) is x movement,
& (deltaPoint y) is y movement of the Plot Window.
WARNING: There is no bounds checking for this, so make sure you keep
the Plot Window visible!
setLineType: bitPattern
Change the type of the line to plot with to the given bitPattern value.
(example: 2r11110000111100001111000011110000 = 16rF0F0F0F0 will draw
a dashed line). This is equivalent to SetDrPt() in graphics.library.
drawText: text at: startPoint
Place the given text at the given starting point using the current
pen colors.
WARNING: There is no bounds checking for this, so make sure you keep
the text inside the Plot Window!
drawBox: fromPoint to: endPoint
Draw a box (fromPoint x) @ (fromPoint y)
to (endPoint x) @ (endPoint y). This is different from the
graphics.library DrawBox() call in that the endPoint is NOT interpreted
to be the width & height of the box. If you want to use the second
point as width @ height, simply add this:
endPoint x <- fromPoint x + endPoint x.
endPoint y <- fromPoint y + endPoint y.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
drawCircleAt: centerPoint radius: r
Draw a circle at the given centerPoint with the given radius using
the current pen colors.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
circleRadius: radius
Draw a circle at the current location, with the given radius using
the current pen colors.

WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
drawTo: endPoint
Draw a line from the current location to the given endPoint using the
current pen colors.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
goTo: aPoint
Move the drawing point to the given aPoint.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
drawLine: fromPoint to: endPoint
Draw a line fromPoint to endPoint using the current pen colors.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
drawPoint: atPoint
Draw a pixel atPoint using the current pen colors.
WARNING: There is no bounds checking for this, so make sure you keep
inside the Plot Window!
direction
This method returns a Radian value, indicating the current direction
that the Pen will go with the go: method.
direction: radianAngle
Set the direction that the Pen will go with the go: method.
erase
Fill the Plot Window with the background color & erase all Plotting.
extent
Return a Point that indicates the width @ height of the Plot Window.
location
Return a Point that indicates the x @ y of the
plotter's location.
center
Move the current plotting location to the center of the Plot Window.
tellPens
Return a Point that indicates the fpen @ bpen
of the Plot Window.
setPens: penSet
Change the fpen @ bpen values to (penSet x) @ (penSet y) respectively.
go: anAmount

Move the plotting location anAmount in the current direction.

anAmount is a scalar value (Integer or Float).

turn: addedAngle

Change the current direction by the given addedAngle (in Radians).

titleIs

Return a String that corresponds to the title of the plot window.

SEE ALSO FormPen , SavePen , ShowPen

1.3 FormPen Class:

The class FormPen is a sub-class of Pen that allows the User

to put together a collection (actually a Bag) of lines.

Responds to

new

Initialize the FormPen class instance.

add: startingPoint to: endPoint

Add a line with the given points to the instance.

with: aPen displayAt: location

Draw all the lines contained in the FormPen using the given aPen.

aPen is of class Pen .

1.4 SavePen Class:

The class SavePen is a sub-class of FormPen that allows the User

to save a drawing made by a Pen. What the original author of

this class means by save isn't quite clear.

Responds to

setForm: aForm

Initialize the instance variable with aForm of class Form .

goTo: aPoint

Add a line from the current location to aPoint of class Point

to aForm.

1.5 ShowPen Class:

The class ShowPen is a sub-class of Pen that allows the User

to see some fancy uses of the Pen class.

Responds to

withPen: aPen

Initialize the instance variable(s) (aPen is of class Pen .

poly: nSides length: length

Draw a ploygon with the given number of sides each with the given

length.

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window! Also, there is no such thing as a

ploygon with less than 3 sides, but this method doesn't

perform any check for this!

spiral: n angle: a

Draw a spiral with the given number of segments (which is also the

length of the segments), changing the direction angle by a Radians .

WARNING: There is no bounds checking for this, so make sure you keep

inside the Plot Window!

1.6 Form Class:

The class Form is a sub-class of Object that allows the

User to draw figures using ASCII text. This class is NOT ported to

the graphic capabilities of the Amiga, so don't expect to get any useful

pictures with it. I've just left the Smalltalk code as descriptions

of what the methods actually do. Use class Pen or the Curses

primitives (in AmigaTalk:User/Curses.st) for drawing simple

pictures instead.

Responds to

new

Initialize the instance of Form.

clipFrom: upperLeft to: lowerRight

"You figure it out:"

! newForm newRow rsize left top rText !

left <- upperLeft y - 1. " left hand side"

top <- upperLeft x - 1.

rsize <- lowerRight y - left.

newForm <- Form new.

(upperLeft x to: lowerRight x)

do: [:i |

newRow <- String new: rsize.

rText <- self row: i.

(1 to: rsize)

do: [:j |

```
newRow at: j
put: (rText at: (left + j)
ifAbsent: [$ ])
].
newForm row: (i - top) put: newRow
].
^ newForm
columns
^ text inject: 0 into: [:x :y | x max: y size ]
display
smalltalk clearScreen.
self printAt: 1 @ 1.
' ' printAt: 20 @ 0
eraseAt: aPoint ! location !
location <- aPoint copy.
text do: [:x | (String new: (x size)) printAt: location.
location x: (location x + 1) ]
extent
^ self rows @ self columns
first
^ text first
next
^ text next
overLayForm: sourceForm at: startingPoint
! newRowNum rowText left rowSize !
newRowNum <- startingPoint x.
left <- startingPoint y - 1.
sourceForm do: [:sourceRow |
rowText <- self row: newRowNum.
rowSize <- sourceRow size.
rowText <- rowText padTo: (left + rowSize).
(1 to: rowSize) do: [:i |
((sourceRow at: i) ~= $ )
ifTrue: [ rowText at: (left + i)
put: (sourceRow at: i)]].
self row: newRowNum put: rowText.
newRowNum <- newRowNum + 1]
placeForm: sourceForm at: startingPoint
! newRowNum rowText left rowSize !
```

```
newRowNum <- startingPoint x.
left <- startingPoint y - 1.
sourceForm do: [:sourceRow |
rowText <- self row: newRowNum.
rowSize <- sourceRow size.
rowText <- rowText padTo: (left + rowSize).
(1 to: rowSize) do: [:i |
rowText at: (left + i)
put: (sourceRow at: i)].
self row: newRowNum put: rowText.
newRowNum <- newRowNum + 1]
reversed ! newForm columns newRow !
columns <- self columns.
newForm <- Form new.
(1 to: self rows) do: [:i |
newRow <- text at: i.
newRow <- newRow ,
(String new: (columns - newRow size)).
newForm row: i put: newRow reversed ].
^ newForm
rotated ! newForm rows newRow !
rows <- self rows.
newForm <- Form new.
(1 to: self columns) do: [:i |
newRow <- String new: rows.
(1 to: rows) do: [:j |
newRow at: ((rows - j) + 1)
put: ((text at: j)
at: i ifAbsent: [$ ])].
newForm row: i put: newRow ].
^ newForm
row: index
^ text at: index ifAbsent: [”]
row: index put: aString
(index > text size)
ifTrue: [ [text size < index] whileTrue:
[text <- text grow: ”] ].
text at: index put: aString
rows
```

ˆ text size
printAt: aPoint ! location !
location <- aPoint copy.
text do: [:x | x printAt: location.
location x: ((location x) + 1)]

1.7 Object Class:

The class Object is a superclass of all classes in the system, and is
used to provide a consistent basic functionality and default behavior.
Many methods in class Object are overridden in subclasses.
Responds to
== Return true if receiver and argument are the
same object, false otherwise.
~~ Inverse of ==.
asString Return a string representation of the receiver, by default
this is the same as printString, although one or the
other is redefined in many subclasses.
asSymbol Return a symbol representing the receiver.
class Return object representing the class of the receiver.
copy Return shallowCopy of receiver. Many subclasses
redefine shallowCopy.
deepCopy Return the receiver. This method is redefined in many
subclasses.
d do: The argument must be a one argument block. Execute the
block on every element of the receiver collection.
Elements in the receiver collection are listed using first
and next, so the default behavior is
merely to execute the block using the receiver as argument.
error: Argument must be a String. Print argument string as
error message. Return nil.
n first Return first item in sequence, which is by
default simply the receiver. See next,
below.
isKindOf: Argument must be a Class. Return true if class of receiver,
or any superclass thereof, is the same as argument.
isMemberOf: Argument must be a Class. Return true if receiver is
instance of argument class.
isNil Test whether receiver is object nil.

n next Return next item in sequence, which is by
default nil. This message is redefined in
classes which represent sequences, such as
Array or Dictionary.
notNil Test if receiver is not object nil.
print Display print image of receiver on the Status Window.
printString Return a string representation of receiver. Objects
which do not redefine printString, and which therefore
do not have a printable representation, return their
class name as a string.
respondsTo: Argument must be a symbol. Return true if receiver will
respond to the indicated message.
shallowCopy Return the receiver. This method is redefined
in many subclasses.
subclassResponsibility:
Inform the user that a subclass did NOT implement the
given method.
notImplemented:
Inform the user that the given method is NOT implemented.
doesNotUnderstand:
Inform the user that a subclass does NOT understand the
given method.
shouldNotImplement:
Inform the user that a subclass should NOT implement the
given method.
Examples: Printed result:
7 ~~ 7.0 True
7 asSymbol #7
7 class Integer
7 copy 7
7 isKindOf: Number True
7 isMemberOf: Number False
7 isNil False
7 respondsTo: #+ True

1.8 UndefinedObject Class:

The pseudo variable nil is an instance (usually the only instance)
of the class UndefinedObject. nil is used to represent undefined
values, and is also typically returned in error situations. nil is also
used as a terminator in sequences, as for example in response to the
message next when there are no further elements in a sequence.
Responds to
r isNil Overrides method found in Object. Return true.
r notNil Overrides method found in Object. Return false.
r printString Return 'nil'.
Examples: Printed result:
nil isNil True

1.9 Symbol Class:

Instances of the class Symbol are created either by their literal
representation, which is a pound sign followed by a string of nonspace
characters (for example #aSymbol), or by the message asSymbol being
passed to an object. Symbols cannot be created using new. Symbols
are guaranteed to have unique representations; that is, two symbols
representing the same characters will always test equal to each other.
Inside of literal arrays, the leading pound signs on symbols can be
eliminated, for example: #(these are symbols).
Responds to
r == Return true if the two symbols represent the
same characters, false otherwise.
r asString Return a String representation of the symbol
without the leading pound sign.
r printString Return a String representation of the symbol,
including the leading pound sign.
Examples: Printed result:
#abc == #abc True
#abc == #ABC False
#abc ~~ #ABC True
#abc printString #abc
'abc' asSymbol #abc

1.10 Boolean Class:

The class Boolean provides protocol for manipulating true and false values. The pseudo-variables true and false are instances of the subclasses of Boolean; True and False, respectively. The subclasses True and False, in combination with blocks, are used to implement conditional control structures. Note, however, that the bytecodes may optimize conditional tests by generating code in-line, rather than using message passing. Note that bit-wise boolean operations are provided by class Integer.

Responds To

& The argument must be a boolean. Return the logical conjunction (and) of the two values.

| The argument must be a boolean. Return the logical disjunction (or) of the two values.

and: The argument must be a block. Return the logical conjunction (and) of the two values. If the receiver is false the second argument is not used, otherwise the result is the value yielded in evaluating the argument block.

or: The argument must be a block. Return the logical disjunction (or) of the two values. If the receiver is true the second argument is not used, otherwise the result is the value yielded in evaluating the argument block.

eqv: The argument must be a boolean. Return the logical equivalence (eqv) of the two values.

xor: The argument must be a boolean. Return the logical exclusive or (xor) of the two values.

Examples: Printed result:

(1 > 3) & (2 < 4) False

(1 > 3) | (2 < 4) True

(1 > 3) and: [2 < 4] False

1.11 True Class:

The pseudo-variable true is an instance (usually the only instance) of
the class True.
Responds To
ifTrue: Return the result of evaluating the argument block.
ifFalse: Return nil.
ifTrue:ifFalse:
Return the result of evaluating the first
argument block.
ifFalse:ifTrue:
Return the result of evaluating the second
argument block.
not Return false.
Examples: Printed result:
(3 < 5) not False
(3 < 5) ifTrue: [17] 17

1.12 False Class:

The pseudo-variable false is an instance (usually the only instance) of
the class False.
Responds To
ifTrue: Return nil.
ifFalse: Return the result of evaluating the argument block.
ifTrue:ifFalse:
Return the result of evaluating the second
argument block.
ifFalse:ifTrue:
Return the result of evaluating the first
argument block.
not Return true.
Examples: Printed result:
(1 < 3) ifTrue: [17] 17
(1 < 3) ifFalse: [17] nil

1.13 Magnitude Class:

The class Magnitude provides protocol for those subclasses possessing a linear ordering. For the sake of efficiency, most subclasses redefine some or all of the relational messages. All methods are defined in terms of the basic messages <, = and >, which are in turn defined circularly in terms of each other. Thus each subclass of Magnitude must redefine at least one of these messages.

Responds To

< Relational less than test. Returns a boolean.

<= Relational less than or equal test.

= Relational equal test. Note that this differs from ==, which is an object equality test.

~= Relational not equal test, opposite of =.

>= Relational greater than or equal test.

> Relational greater than test.

between:and: Relational test for inclusion.

max: Return the maximum of the receiver and argument value.

min: Return the minimum of the receiver and argument value.

Examples: Printed result:

$A max: $a $a

4 between: 3.1 and: (17/3) True

1.14 Char Class:

This class defines protocol for objects with character values. Characters possess an ordering given by the underlying representation, however arithmetic is not defined for character values. Characters are written literally by preceding the character desired with a dollar sign, for example: $a $B $$.

Responds To

r == Object equality test. Two instances of the same character always test equal.

asciiValue Return an Integer representing the ASCII value of the receiver.

asLowercase If the receiver is an uppercase letter returns the same letter in lowercase, otherwise returns the receiver.

asUppercase If the receiver is a lowercase letter returns

the same letter in uppercase, otherwise
returns the receiver.
r asString Return a length one string containing the
receiver. Does not contain leading dollar
sign, compare to printString.
digitValue If the receiver represents a number (for
example $9) return the digit value of the
number. If the receiver is an uppercase
letter (for example $B) return the position
of the number in the uppercase letters + 10,
($B returns 11, for example). If the
receiver is neither a digit nor an uppercase
letter an error is given and nil returned.
isAlphaNumeric
Respond true if receiver is either digit or
letter, false otherwise.
isDigit Respond true if receiver is a digit, false otherwise.
isLetter Respond true if receiver is a letter, false otherwise.
isLowercase Respond true if receiver is a lowercase letter, false
otherwise.
isSeparator Respond true if receiver is a space, tab or
newline, false otherwise.
isUppercase Respond true if receiver is an uppercase letter, false
otherwise.
isVowel Respond true if receiver is $a, $e, $i, $o or
$u, in either upper or lower case.
r printString Respond with a string representation of the
character value. Includes leading dollar
sign, compare to asString, which does not
include $.
Examples: Printed result:
$A < $0 False
$A asciiValue 65
$A asString A
$A printString $A
$A isVowel True
$A digitValue 10

1.15 Number Class:

The class Number is an abstract superclass for Integer and Float.
Instances of Number cannot be created directly. Relational messages
and many arithmetic messages are redefined in each subclass for arguments
of the appropriate type. In general, an error message is given and nil
returned for illegal arguments.
Responds To
+ Mixed type addition.
- Mixed type subtraction.
* Mixed type multiplication
/ Mixed type division.
n ^ Exponentiation, same as raisedTo:.
@ Construct a point with coordinates being the
receiver and the argument.
abs Absolute value of the receiver.
exp e raised to the power.
n gamma Return the gamma function (generalized fac-
torial) evaluated at the receiver.
ln Natural logarithm of the receiver.
log: Logarithm in the given base.
negated The arithmetic inverse of the receiver.
negative True if the receiver is negative.
n pi Return the approximate value of the receiver
multiplied by (3.1415926...).
positive True if the receiver is positive.
n radians Argument converted into radians.
raisedTo: The receiver raised to the argument value.
reciprocal The arithmetic reciprocal of the receiver.
roundTo: The receiver rounded to units of the argument.
sign Return -1, 0 or 1 depending upon whether the
receiver is negative, zero or positive.
sqrt Square root. nil if receiver is less than
zero.
squared Return the receiver multiplied by itself.
strictlyPositive
True if the receiver is greater than zero.
to: Interval from receiver to argument value with
step of 1.

to:by: Interval from receiver to argument in given
steps.
truncatedTo: The receiver truncated to units of the argument.
Examples: Printed result:
3 < 4.1 True
3 + 4.1 7.1
3.14159 exp 23.1406
9 gamma 40320
5 reciprocal 0.2
0.5 radians 0.5 radians
13 roundTo: 5 15
13 truncateTo: 5 10

1.16 Integer Class:

The class Integer provides protocol for objects with integer values.
Responds To
r == Object equality test. Two integers representing the
same value are considered to be the same object.
// Integer quotient, truncated towards negative
infinity (compare to quo:).
\ Integer remainder, truncated towards negative
infinity (compare to rem:).
allMask: Argument must be Integer. Treating receiver
and argument as bit strings, return true if
all bits with 1 value in argument correspond
to bits with 1 values in the receiver.
anyMask: Argument must be Integer. Treating receiver
and argument as bit strings, return true if
any bit with 1 value in argument corresponds
to a bit with 1 value in the receiver.
asCharacter Return the Char with the same underlying
ASCII representation as the low order eight
bits of the receiver.
asFloat Floating point value with same magnitude as
receiver.
bitAnd: Argument must be Integer. Treating the
receiver and argument as bit strings, return
logical and of values.

bitAt: Argument must be Integer greater than 0 and
less than underlying word size. Treating
receiver as a bit string, return the bit value at the
given position, numbering from low order (or rightmost)
position.
bitInvert Return the receiver with all bit positions inverted.
bitOr: Return logical or of values.
bitShift: Treating the receiver as a bit string, shift
bit values by amount indicated in argument.
Negative values shift right, positive left.
bitXor: Return logical xor of values.
even Return true if receiver is even, false otherwise.
factorial Return the factorial of the receiver. Return
as Float for large numbers.
gcd: Argument must be Integer. Return the
greatest common divisor of the receiver and
argument.
highBit Return the location of the highest 1 bit in
the receiver. Return nil for receiver zero.
lcm: Argument must be Integer. Return least com-
mon multiple of receiver and argument.
noMask: Argument must be Integer. Treating receiver
and argument as bit strings, return true if
no 1 bit in the argument corresponds to a 1
bit in the receiver.
odd Return true if receiver is odd, false otherwise.
quo: Return quotient of receiver divided by argument.
radix: Return a string representation of the receiver value,
printed in the base represented by the argument.
Argument value must be less than 36.
rem: Remainder after receiver is divided by argument value.
timesRepeat: Repeat argument block the number of times given by
the receiver.
Examples: Printed result:
5 + 4 7
5 allMask: 4 True
4 allMask: 5 False
5 anyMask: 4 True
5 bitAnd: 3 1

5 bitOr: 3 7

5 bitInvert -6

254 radix: 16 16rFE

-5 // 4 -2

-5 quo: 4 -1

-5 \ 4 1

-5 rem: 4 -1

8 factorial 40320

1.17 Float Class:

The class Float provides protocol for objects with floating point values.

Responds To

r == Object equality test. Return true if the

receiver and argument represent the same

floating point value.

n ^ Floating exponentiation.

arcCos Return a Radian representing the arcCos of

the receiver.

arcSin Return a Radian representing the arcSin of

the receiver.

arcTan Return a Radian representing the arcTan of

the receiver.

asFloat Return the receiver.

ceiling Return the Integer ceiling of the receiver.

coerce: Coerce the argument into being type Float.

exp Return e raised to the receiver value.

floor Return the Integer floor of the receiver.

fractionPart Return the fractional part of the receiver.

n gamma Return the value of the gamma function applied to

the receiver value.

integerPart Return the integer part of the receiver.

ln Return the natural log of the receiver.

radix: Return a string containing the printable representation

of the receiver in the given radix. Argument must be

an Integer less than 36.

rounded Return the receiver rounded to the nearest integer.

sqrt Return the square root of the receiver.

truncated Return the receiver truncated to the nearest integer.

Examples: Printed result:
4.2 * 3 12.6
2.1 |^ 4 19.4481
2.1 raisedTo: 4 19.4481
0.5 arcSin 0.523599 radians
2.1 reciprocal 0.47619
4.3 sqrt 2.07364

1.18 Radian Class:

The class Radian is used to represent radians. Radians are a unit of measurement, independent of other numbers. Only radians will respond to the trigonometric functions such as sin & cos. Numbers can be converted into radians by passing them the message radians. Similarly, radians can be converted into numbers by sending them the message asFloat. Notice that only a limited range of arithmetic operations are permitted on Radians. Radians are normalized to be between 0 and 2 * pi .
Responds To
+ Argument must be a Radian. Add the two radians together and return the normalized result.
- Argument must be a Radian. Subtract the argument from the receiver and return the normalized result.
* Argument must be a Number. Multiply the receiver by the argument amount and return the normalized result.
/ Argument must be a Number. Divide the receiver by the argument amount and return the normalized result.
asFloat Return the receiver as a floating point number.
cos Return a floating point number representing the cosine of the receiver.
sin Return a floating point number representing the sine of the receiver.
tan Return a floating point number representing the tangent of the receiver.
Examples: Printed result:
0.5236 radians sin 0.5
0.5236 radians cos 0.866025
0.5236 radians tan 0.577352
0.5 arcSin asFloat 0.523599

1.19 Point Class:

Points are used to represent pairs of quantities, such
as coordinate pairs.
Responds To
< True if both values of the receiver are less
than the corresponding values in the argument.
<= True if the first value is less than or equal
to the corresponding value in the argument,
and the second value is less than the
corresponding value in the argument.
>= True if both values of the receiver are
greater than or equal to the corresponding
values in the argument.
* Return a new point with coordinates multi-
plied by the argument value.
/ Return a new point with coordinates divided
by the argument value.
// Return a new point with coordinates divided
by the argument value.
+ Return a new point with coordinates offset by
the corresponding values in the argument.
abs Return a new point with coordinates having
the absolute value of the receiver.
dist: Return the Euclidean distance between the
receiver and the argument point.
max: The argument must be a Point. Return the
lower right corner of the rectangle defined
by the receiver and the argument.
min: The argument must be a Point. Return the
upper left corner of the rectangle defined by
the receiver and the argument.
transpose Return a new point with coordinates being the
transpose of the receiver.
x Return the first coordinate of the receiver.
x: Set the first coordinate of the receiver.
x:y: Sets both coordinates of the receiver.
y Return the second coordinate of the receiver.
y: Set the second coordinate of the receiver.

Examples: Printed result:
(10@12) < (11@14) True
(10@12) < (11@11) False
(10@12) max: (11@11) 11@12
(10@12) min: (11@11) 10@11
(10@12) dist: (11@14) 2.23607
(10@12) transpose 12@10

1.20 Random Class:

The class Random provides protocol for random number generation.
Sending the message next to an instance of Random results in a Float
between 0.0 and 1.0, randomly distributed. By default, the pseudo-random
sequence is the same for each object in class Random. This can be
altered using the message "randomize".
Responds To
n between:and: Return a random number uniformly distributed
between the two arguments.
n first Return a random number between 0.0 and 1.0.
This message merely provides consistency with
protocol for other sequences, such as Arrays
or Intervals.
next Return a random number between 0.0 and 1.0.
d next: Return an Array containing the next n random
numbers, where n is the argument value.
n randInteger: The argument must be an Integer. Return a
random integer between 1 and the value given.
n randomize Change the pseudo-random number generator
seed by a time dependent value.
Examples: Printed result:
i <- Random new
i next 0.759
i next 0.157
i next: 3 #(0.408 0.278 0.547)
i randInteger: 12 5
i between: 4 and: 17.5 10.0

1.21 Collection Class:

The class Collection provides protocol for groups of objects, such as Arrays or Sets. The different forms of collections are distinguished by several characteristics, among them whether the size of the collection is fixed or unbounded, the presence or absence of an ordering, and their insertion or access method. For example, an Array is a collection with a fixed size and ordering, indexed by integer keys. A Dictionary, on the other hand, has no fixed size or ordering, and can be indexed by arbitrary elements. Nevertheless, Arrays and Dictionarys share many features in common, such as their access method (at: and at:put:), and the ability to respond to collect:, select:, and many other messages. The table below lists some of the characteristics of several forms of collections:

Name Creation Size Ordered? Insertion Access
Method fixed? method method

Bag/Set new no no add: includes:
Dictionary new no no at:put: at:
Interval n to: m yes yes none at:
List new no yes addFirst: first
addLast: last
Array new: yes yes at:put: at:
String new: yes yes at:put: at:

The list below shows messages that are shared in common by all collections.
Responds To
addAll: The argument must be a Collection. Add all the elements of the argument collection to the receiver collection.
asArray Return a new collection of type Array containing the elements from the receiver collection. If the receiver was ordered, the elements will be in the same order in the new collection, otherwise the elements will be in an arbitrary order.
asBag Return a new collection of type Bag containing the elements from the receiver collection.

n asList Return a new collection of type List containing the
elements from the receiver collection. If the receiver
was ordered, the elements will be in the same order in
the new collection, otherwise the elements will be in
an arbitrary order.
asSet Return a new collection of type Set containing the
elements from the receiver collection.
asString Return a new collection of type String containing the
elements from the receiver collection. The elements to
be included must all be of type Character. If the
receiver was ordered, the elements will be in the same
order in the new collection, otherwise the elements will
be listed in an arbitrary order.
coerce: The argument must be a Collection. Return a collection,
of the same type as the receiver, containing elements
from the argument collection. This message is redefined
in most subclasses of Collection.
collect: The argument must be a one argument block. Return a new
collection, like the receiver, containing the result of
evaluating the argument block on each element of the
receiver collection.
detect: The argument must be a one argument block. Return the
first element in the receiver collection for which the
argument block evaluates true. Report an error and
return "nil" if no such element exists. Note that in
unordered collections (such as Bags or Dictionarys) the
first element to be encountered that will satisfy the
condition may not be easily predictable.
detect:ifAbsent:
Return the first element in the receiver collection for
which the first argument block evaluates true. Return
the result of evaluating the second argument if no such
element exists.
do: The argument must be a one argument block. Evaluate the
argument block on each element in the receiver collection.
includes: Return true if the receiver collection contains the
argument.
inject:into: The first argument must be a value, the second a two
argument block. The second argument is evaluated once

for each element in the receiver collection, passing as
arguments the result of the previous evaluation (starting
with the first argument) and the element. The value
returned is the final value generated.
isEmpty Return true if the receiver collection contains no
elements.
occurrencesOf:
Return the number of times the argument occurs in the
receiver collection.
remove: Remove the argument from the receiver collection. Report
an error if the element is not contained in the receiver
collection.
remove:ifAbsent:
Remove the first argument from the receiver collection.
Evaluate the second argument if not present.
reject: The argument must be a one argument block. Return a new
collection like the receiver containing all elements for
which the argument block returns false.
select: The argument must be a one argument block. Return a new
collection like the receiver containing all elements for
which the argument block returns true.
size Return the number of elements in the receiver
collection.
Examples: Printed result:
i <- 'abacadabra'
i size 10
i asArray #($a $b $a $c $a $d $a $b $r $a)
i asBag Bag ($a $a $a $a $a $r $b $b $c $d)
i asSet Set ($a $r $b $c $d)
i occurrencesOf: $a 5
i reject: [:x | x isVowel] bcdbr

1.22 Bags & Sets Classes:

Bags and Sets are each unordered collections of elements. Elements in
the collections do not have keys, but are added and removed directly.
The difference between a Bag and a Set is that each element can occur
any number of times in a Bag, whereas only one copy is inserted into
a Set.

Responds To
add: Add the indicated element to the receiver collection.
add:withOccurences:
(Bag only) Add the indicated element to the
receiver Bag the given number of times.
n first Return the first element from the receiver collection.
As the collection is unordered, the first element depends
upon certain values in the internal representation, and is
not guaranteed to be any specific element in the
collection.
n next Return the next element in the collection. In conjunction
with first, this can be used to access each element of
the collection in turn.
Examples: Printed result:
i <- (1 to: 6) asBag Bag (1 2 3 4 5 6)
i size 6
i select: [:x | (x \ 2) strictlyPositive] Bag (1 3 5)
i collect: [:x | x \ 3] Bag (0 0 1 1 2 2)
j <- (i collect: [:x | x \ 3]) asSet Set (0 1 2)
j size 3
Note: Since Bags and Sets are unordered, there is no way to
establish a mapping between the elements of the Bag i in the
example above and the corresponding elements in the collection that
resulted from the message collect: [:x | x \ 3].

1.23 KeyedCollection Class:

The class KeyedCollection provides protocol for collections with keys,
such as Dictionarys and Arrays. Since each entry in the collection has
both a key and value, the method add: is no longer appropriate. Instead,
the method at:put:, which provides both a key and a value, must be used.
Responds To
asDictionary Return a new collection of type Dictionary
containing the elements from the receiver
collection.
at: Return the item in the receiver collection
whose key matches the argument. Produces and
error message, and returns nil, if no item is
currently in the receiver collection under

the given key.
at:ifAbsent: Return the element stored in the dictionary
under the key given by the first argument.
Return the result of evaluating the second
argument if no such element exists.
atAll:put: The first argument must be a collection con-
taining keys valid for the receiver. At each
location given by a key in the first argument
place the second argument.
binaryDo: The argument must be a two argument block.
This message is similar to do:, however both
the key and the element value are passed as
argument to the block.
includesKey: Return true if the indicated key is valid for
the receiver collection.
indexOf: Return the key value of the first element in
the receiver collection matching the argument.
Produces an error message if no such element exists.
Note that, as with the message detect:, in unordered
collections the first element may not be related in any
way to the order in which elements were placed into the
collection, but is rather implementation dependent.
indexOf:ifAbsent:
Return the key value of the first element in the receiver
collection matching the argument. Return the result of
evaluating the second argument if no such element exists.
keys Return a Set containing the keys for the
receiver collection.
keysDo: The argument must be a one argument block.
Similar to do:, except that the values passed
to the block are the keys of the receiver
collection.
keysSelect: Similar to select, except that the selection
is made on the basis of keys instead of
values.
removeKey: Remove the object with the given key from the
receiver collection. Print an error message,
and return nil, if no such object exists.
Return the value of the deleted item.

removeKey:ifAbsent:
Remove the object with the given key from the
receiver collection. Return the result of
evaluating the second argument if no such
object exists.
values Return a Bag containing the values from the
receiver collection.
Examples: Printed result:
i <- 'abacadabra'
i atAll: (1 to: 7 by: 2) put: $e ebecedebra
i indexOf: $r 9
i atAll: i keys put: $z zzzzzzzzzz
i keys Set (1 2 3 4 5 6 7 8 9 10)
i values Bag ($z $z $z $z $z $z $z $z $z $z)
#(how odd) asDictionary Dictionary (1 @ #how 2 @ odd)

1.24 Dictionary Class:

A Dictionary is an unordered collection of elements, as are Bags and
Sets. However, unlike these collections, elements inserted and removed
from a Dictionary must reference an explicit key. Both the key and
value portions of an element can be any object, although commonly the
keys are instances of Symbol or Number.
Responds To
at:put: Place the second argument into the receiver collection
under the key given by the first argument.
currentKey Return the key of the last element yielded in
response to a first or next request.
n first Return the first element of the receiver collection.
Return nil if the receiver collection is empty.
n next Return the next element of the receiver collection, or
nil if no such element exists.
Examples: Printed result:
i <- Dictionary new
i at: #abc put: #def
i at: #pqr put: #tus
i at: #xyz put: #wrt
i print Dictionary (#abc @ #def #pqr @ #tus #xyz @ #wrt)
i size 3
i at: #pqr #tus
i indexOf: #tus #pqr
i keys Set (#abc #pqr #xyz)
i values Bag (#wrt #def # tus)

1.25 AmigaTalk Class:

The class AmigaTalk provides protocol for the pseudo-variable amigatalk.
Since it is a subclass of Dictionary, this variable can be used to store
information, and thus provide a means of communication between objects.
Other messages modify various parameters used by the AmigaTalk system.
Responds To
n date Return the current date and time as a string.
n display Set execution display to display the result of every
expression typed, but not for assignments. Note that the
display behavior can also be modified using the -d
argument on the command line.
n displayAssign
Set execution display to display the result of every
expression typed, including assignment statements.
n doPrimitive:withArguments:
Execute the indicated primitive with arguments given by
the second array. A few primitives (such as those
dealing with process management) cannot be executed in
this manner.
n noDisplay Turn off execution display - no results will
be displayed unless explicitly requested by
the user.
d perform:withArguments:
Send indicated message to the receiver, using
the arguments given. The first value in the
argument array is taken to be the receiver of
the message. Unpredictable results if the
number of arguments is not appropriate for
the given message.
n sh: The argument, which must be a string, is executed as an
A,migaDOS command by the shell. The value returned is
the termination status of the shell.
n time: The argument must be a block. The block is executed,
and the number of seconds elapsed during execution
returned. Time is only accurate to within about one
second.
Examples: Printed result:
amigatalk date Fri Apr 12 16:15:42 1985
amigatalk perform: #+ withArguments: #(2 5) 7
amigatalk doPrimitive: 10 withArguments: #(2 5) 7

1.26 SequenceableCollection Class:

The class SequenceableCollection contains protocol for collections
that have a definite sequential ordering and are indexed by integer
keys. Since there is a fixed order for elements, it is possible to
refer to the last element in a SequenceableCollection.
Responds To
, Appends the argument collection to the receiver
collection, returning a new collection of the same
type as the receiver.
copyFrom:to: Return a new collection, like the receiver,
containing the designated subportion of the
receiver collection.
copyWith: Return a new collection, like the receiver,
with the argument added to the end.
copyWithout: Return a new collection, like the receiver,
with all occurrences of the argument removed.
equals:startingAt:
The first argument must be a SequenceableCol-
lection. Return true if each element of the
receiver collection is equal to the corresponding
element in the argument offset by the amount given
in the second argument.
findFirst: Find the key for the first element whose
value satisfies the argument block. Produce
an error message if no such element exists.
findFirst:ifAbsent:
Both arguments must be blocks. Find the key
for the first element whose value satisfies
the first argument block. If no such element
exists return the value of the second argu-
ment.
findLast: Find the key for the last element whose value
satisfies the argument block. Produce an
error message if no such element exists.
findLast:ifAbsent:
Both arguments must be blocks. Find the key
for the last element whose value satisfies
the first argument block. If no such element

exists return the value of the second
argument block.
firstKey Return the first key valid for the receiver
collection.
indexOfSubCollection:startingAt:
Starting at the position given by the second
argument, find the next block of elements in
the receiver collection which match the col-
lection given by the first argument, and
return the index for the start of that block.
Produce an error message if no such position
exists.
indexOfSubCollection:startingAt:ifAbsent:
Similar to indexOfSubCollection:startingAt:,
except that the result of the exception block
is produced if no position exists matching
the pattern.
last Return the last element in the receiver collection.
lastKey Return the last key valid for the receiver collection.
replaceFrom:to:with:
Replace the elements in the receiver collec-
tion in the positions indicated by the first
two arguments with values taken from the col-
lection given by the third argument.
replaceFrom:to:with:startingAt:
Replace the elements in the receiver collec-
tion in the positions indicated by the first
two arguments with values taken from the col-
lection given in the third argument, starting
at the position given by the fourth argument.
n reversed Return a collection, like the receiver, with
elements reversed.
reverseDo: Similar to do:, except that the items are
presented in reverse order.
n sort Return a collection, like the receiver, with
the elements sorted using the comparison <=.
Elements must be able to respond to the
binary message <=.
n sort: The argument must be a two argument block

which yields a boolean. Return a collection,
like the receiver, sorted using the argument
to compare elements for the purpose of
ordering.
with:do: The second argument must be a two argument
block. Present one element from the receiver
collection and from the collection given by
the first argument in turn to the second
argument block. An error message is given if
the collections do not have the same number
of elements.
Examples: Printed result:
i <- 'abacadabra'
i copyFrom: 4 to: 8 cadab
i copyWith: $z abacadabraz
i copyWithout: $a bcdbr
i findFirst: [:x | x > $m] 9
i indexOfSubCollection: 'dab' startingAt: 16
i reversed arbadacaba
i , i reversed abacadabraarbadacaba
i sort: [:x :y | x >= y] rdcbbaa

1.27 Interval Class:

The class Interval represents a sequence of numbers in an arithmetic
sequence, either ascending or descending. Instances of Interval are
created by numbers in response to the message to: or to:by:. In
conjunction with the message do:, Intervals create a control structure
similar to do or for loops in Algol-like languages. For example:
(1 to: 10) do: [:x | x print]
will print the numbers 1 through 10. Although they are a collection,
Intervals cannot be added to. They can, however, be accessed randomly
using the message at:.
Responds To
first Produce the first element from the interval.
In conjunction with "last", this message may be
used to produce each element from the inter-
val in turn. Note that Intervals also
respond to the message "at:", which can be used

to produce elements in an arbitrary order.
from:to:by: Initialize the upper and lower bounds and the
step size for the receiver. (This is used
principally internally by the method for
number to create new Intervals).
next Produce the next element from the interval.
size Return the number of elements that will be
generated in producing the interval.
Examples: Printed result:
(7 to: 13 by: 3) asArray #(7 10 13)
(7 to: 13 by: 3) at: 2 10
(1 to: 10) inject: 0 into: [:x :y | x + y] 55
(7 to: 13) copyFrom: 2 to: 5 #(8 9 10 11)
(3 to: 5) copyWith: 13 #(3 4 5 13)
(3 to: 5) copyWithout: 4 #(3 5)
(2 to: 4) equals: (1 to: 4) startingAt: 2 True

1.28 LinkedList Class:

Lists represent collections with a fixed order, but indefinite size.
No keys are used, and elements are added or removed from one end of
the other. Used in this way, Lists can perform as stacks or as
queues. The table below illustrates how stack and queue operations
can be implemented in terms of messages to instances of List.
stack operations queue operations

---

push addLast: add addLast:
pop removeLast first in queue first
top last remove first in queue removeFirst
test empty isEmpty test empty isEmpty
Responds To
add: Add the element to the beginning of the receiver
collection. This is the same as addFirst:.
addAllFirst: The argument must be a SequenceableCollection. The
elements of the argument are added, in order, to the
front of the receiver collection.
addAllLast: The argument must be a SequenceableCollection. The
elements of the argument are added, in order, to the end
of the receiver collection.

addFirst: The argument is added to the front of the receiver
collection.
addLast: The argument is added to the back of the receiver
collection.
removeFirst Remove the first element from the receiver collection,
returning the removed value.
removeLast Remove the last element from the receiver collection,
returning the removed value.
Examples: Printed result:
i <- List new
i addFirst: 2 / 3 List (0.6666)
i add: $A
i addAllLast: (12 to: 14 by: 2)
i print List (0.6666 $A 12 14)
i first 0.6666
i removeLast 14
i print List (0.6666 $A 12)

1.29 Semaphore Class:

Semaphores are used to synchronize concurrently running Processes.
Responds To
new: If created using new, a Semaphore starts out with zero
excess signals. Alternatively, a Semaphore can be created
with an arbitrary number of excess signals by giving it an
argument to new:.
signal If there is a process blocked on the semaphore it is
scheduled for execution, otherwise the number of excess
signals is incremented by one.
wait If there are excess signals associated with the semaphore
the number of signals is decremented by one, otherwise
the current process is placed on the semaphore queue.

1.30 File Class:

A File is a type of collection where the elements of the collection are
stored on an external medium, typically a disk. For this reason,
although most operations on collections are defined for files, many can
be quite slow in execution. A file can be opened in one of three

modes: In character mode every read returns a single character from
the file. In integer mode every read returns a single word, as an
integer value. In string mode every read returns a single line, as a
String. For writing, character and string modes will write the string
representation of the argument, while integer mode must write only a
single integer.
Responds To
at: Return the object stored at the indicated position.
Position is given as a character count from the start
of the file.
at:put: Place the object at the indicated position in the file.
Position is given as a character count from the start
of the file.
characterMode
Set the mode of the receiver file to character.
currentKey Return the current position in the file, as a
character count from the start of the file.
integerMode Set the mode of the receiver file to integer.
open: Open the indicated file for reading. The argument must
be a String.
open:for: The for: argument must be one of r, w or r+ (see
fopen(3) in the Unix programmers manual). Open the file
in the indicated mode.
read Return the next object from the file.
size Return the size of the file, in character counts.
stringMode Set the mode of the receiver file to string.
write: Write the argument into the file.

1.31 ArrayedCollection Class:

The class ArrayedCollection provides protocol for collections with a
fixed size and integer keys. Unlike other collections, which are
created using the message new, instances of ArrayedCollection must be
created using the one argument message new:. The argument given with
this message must be a positive integer, representing the size of the
collection to be created. In addition to the protocol shown, many of
the methods inherited from superclasses are redefined in this class.
Responds To
= The argument must also be an Array. Test whether the

receiver and the argument have equal elements listed
in the same order.
at:ifAbsent: Return the element stored with the given key. Return the
result of evaluating the second argument if the key is not
valid for the receiver collection.
n padTo: Return an array like the received that is at least as long
as the argument value. Returns the receiver if it is
already longer than the argument.
Examples: Printed result:
'small' = 'small' True
'small' = 'SMALL' False
'small' asArray #($s $m $a $l $l)
'small' asArray = 'small' True
#(1 2 3) padTo: 5 #(1 2 3 nil nil)
#(1 2 3) padTo: 2 #(1 2 3)

1.32 Array Class:

Instances of the class Array are perhaps the most commonly used data
structure in Smalltalk programs. Arrays are represented textually by
a pound sign preceding the list of array elements.
Responds To
at: Return the item stored in the position given by the
argument. An error message is produced, and nil
returned, if the argument is not a valid key.
at:put: Store the second argument in the position given by the
first argument. An error message is produced, and nil
returned, if the argument is not a valid key.
grow: Return a new array one element larger than the receiver,
with the argument value attached to the end. This is a
slightly more efficient command than copyWith:, although
the effect is the same.
Examples: Printed result:
i <- #(110 101 97)
i size 3
i <- i grow: 116 #(110 101 97 116)
i <- i collect: [:x | x asCharacter] #(#n #e #a #t)
i asString neat

1.33 ByteArray Class:

A ByteArray is a special form of array in which the elements must be numbers in the range 0-255. Instances of ByteArray are given a very compact encoding, and are used extensively internally in the AmigaTalk system. A ByteArray can be represented textually by a pound sign preceding the list of array elements surrounded by a pair of square braces.

Responds To

at: Return the item stored in the position given by the argument. An error message is produced, and nil returned, if the argument is not a valid key.

at:put: Store the second argument in the position given by the first argument. An error message is produced, and nil returned, if the argument is not a valid key.

Examples: Printed result:

i <- #[110 101 97]

i size 3

i <- i copyWith: 116 #[110 101 97 116]

i <- i asArray collect: [:x | x asCharacter] #(#n #e #a #t)

i asString neat

1.34 String Class:

Instances of the class String are similar to Arrays, except that the individual elements must be Character. Strings are represented literally by placing single quote marks around the characters making up the string. Strings also differ from Arrays in that Strings possess an ordering, given by the underlying ASCII sequence.

Responds To

, Concatenates the argument to the receiver string, producing a new string. If the argument is not a String it is first converted using printString.

< The argument must be a String. Test if the receiver is lexically less than the argument. For the purposes of comparison, case differences are ignored.

<= Test if the receiver is lexically less than or equal to the argument.

>= Test if the receiver is lexically greater than or equal

to the argument.
> Test if the receiver is lexically greater than the
argument.
r asSymbol Return a Symbol with characters given by the
receiver string.
at: Return the character stored at the position given by the
argument. Produce an error message, and return nil, if
the argument does not represent a valid key.
at:put: Store the character given by second argument at the
location given by the first argument. Produce an error
message, and return nil, if either argument is invalid.
n copyFrom:length:
Return a substring of the receiver. The substring is
taken from the indicated starting position in the receiver
and extends for the given length. Produce an error
message, and return nil, if the given positions are not
legal.
r copyFrom:to: Return a substring of the receiver. The substring is
taken from the indicated positions. Produce an error
message, and return nil, if the given positions are
not legal.
n printAt: The argument must be a Point which describes a location on
the Curses screen. The string is printed at the specified
location.
size Return the number of characters stored in the string.
sameAs: Return true if the receiver and argument string match with
the exception of case differences. Note that the boolean
message =, inherited from ArrayedCollection, can be used
to see if two strings are the same including case
differences.
Examples: Printed result:
'example' at: 2 $x
'bead' at: 1 put: $r read
'small' > 'BIG' True
'small' sameAs: 'SMALL' True
'tary' sort arty
'Rats live on no evil Star' reversed ratS live on no evil staR

1.35 Block Class:

Although it is easy for the programmer to think of blocks as a syntactic construct, or a control structure, they are actually objects, and share attributes of all other objects in the Smalltalk system, such as the ability to respond to messages.

Responds To

fork Start the block executing as a Process. The value nil is immediately returned, and the Process created from the block is scheduled to run in parallel with the current process.

forkWith: Similar to fork, except that the array is passed as arguments to the receiver block prior to scheduling for execution.

newProcess A new Process is created for the block, but is not scheduled for execution.

n newProcessWith:

Similar to newProcess, except that the array is passed as arguments to the receiver block prior to it being made into a process.

value Evaluates the receiver block. Produces an error message, and returns nil, if the receiver block required arguments. Return the value yielded by the block.

value: Evaluates the receiver block. Produces an error message, and returns nil, if the receiver block did not require a single argument. Return the value yielded by the block.

value:value: Two argument block evaluation.

value:value:value:

Three argument block evaluation.

value:value:value:value:

Four argument block evaluation.

value:value:value:value:value:

Five argument block evaluation.

whileTrue: The receiver block is repeatedly evaluated. While it evaluates to true, the argument block is also evaluated. Return nil when the receiver block no longer evaluates to true.

whileTrue The receiver block is repeatedly evaluated until it returns a value that is not true.

whileFalse: The receiver block is repeatedly evaluated. While it
evaluates to false, the argument block is also evaluated.
Return nil when the receiver block no longer evaluates
to false.
whileFalse The receiver block is repeatedly evaluated until it
returns a value that is not false.
Examples: Printed result:
['block indeed'] value block indeed
[:x :y | x + y + 3] value: 5 value: 7 15

1.36 Class Class:

The class Class provides protocol for manipulating class instances. An
instance of class Class is generated for each class in the AmigaTalk
system. New instances of this class are then formed by sending messages
to the class instance.
Responds To
n deepCopy: The argument must be an instance of the receiver class.
A deepCopy of the argument is returned.
n edit The user is placed into a editor editing the file from
which the class description was originally obtained. When
the editor terminates, the class description will be
reparsed and will override the previous description.
See also view.
n list Lists all subclasses of the given class recursively. In
particular, Object list will list the names of all the
classes in the system.
new A new instance of the receiver class is returned. If the
methods for the receiver contain protocol for new, the
new instance will first be passed this message.
new: A new instance of the receiver class is returned. If the
methods for the receiver contain protocol for new:, the
new instance will first be passed this message.
n respondsTo List all the messages that the current class will respond
to.
d respondsTo: The argument must be a Symbol. Return true if the
receiver class, or any of its superclasses, contains a
method for the indicated message. Return false otherwise.
n shallowCopy: The argument must be an instance of the receiver class.

A shallowCopy of the argument is returned.
n superClass Return the superclass of the receiver class.
n variables Return an array containing the names of the instance
variables used in the receiver class.
n view Place the user into an editor viewing the class description from which the class was created. Changes made to
the file will not, however, affect the current class
representation.
getByteArray:
Return a ByteArray that represents the given method in
the given class.
Examples: Printed result:
Array new: 3 #(nil nil nil)
Bag respondsTo: #add: True
SequenceableCollection superClass KeyedCollection

1.37 Process Class:

Processes are created by the system, or by passing the message
newProcess or fork to a block; they cannot be created directly
by the user.
Responds To
block The receiver process is marked as being blocked. This is
usually the result of a semaphore wait. Blocked processes
are not executed.
resume If the receiver process has been suspended, it is
rescheduled for execution.
suspend If the receiver process is scheduled for execution, it is
marked as suspended. Suspended processes are not executed.
state The current state of the receiver process is returned
as a Symbol.
terminate The receiver process is terminated. Unlike a blocked or
suspended process, a terminated process cannot be
restarted.
unblock If the receiver process is currently blocked, it is
scheduled for execution.
yield Returns nil. As a side effect, however, if there are
pending processes, the current process is placed back on
the process queue and another process started.