# Collections

## Protocol Methods

|  |  |
| --- | --- |
| Accessing | size, capacity, at:, at:put: |
| Testing | isEmpty, includes:, contains:, occurrencesOf: |
| Adding | add:, addAll: |
| Removing | remove:, remove:ifAbsent:, removeAll: |
| Enumerating | do:, collect:, select:, reject:, detect:, detect:ifNone:, inject:into: |
| Converting | asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: |
| Creation | with:, with:with:, with:with:with:, with:with:with:with:, withAll: |



# Macros

'look-<t>-here' expandMacros

--> 'look- -here'

'<1s> is <2s>' expandMacrosWith: 'Pharo' with: 'cool'

--> 'Pharo is cool'

'<2s> is <1s>' expandMacrosWith: 'Pharo' with: 'cool'

--> 'cool is Pharo'

'<1p> or <1s>' expandMacrosWith: 'Pharo' with: 'cool'

--> '''Pharo'' or Pharo'

'<1?Quentin:Thibaut> plays' expandMacrosWith: true

--> 'Quentin plays'

'<1?Quentin:Thibaut> plays' expandMacrosWith: false

--> 'Thibaut plays'

# Proxies

Other messages that may be directly interpreted by the VM, depending on the

receiver, include:

class +- < > <= >= = ~= \* / \ == @ bitShift: // bitAnd: bitOr: at: at:put:

size next nextPut: atEnd blockCopy: value value: do: new new: x y.

Selectors that are never sent, because they are inlined by the compiler and

transformed to comparison and jump bytecodes:

ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue: and: or: while-

False: whileTrue: whileFalse whileTrue to:do: to:by:do: caseOf:

caseOf:otherwise: ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:

# Reflection

Reflection refers to the ability to query, examine and even modify the metaobjects

of the runtime system as ordinary objects.

• The Inspector uses instVarAt: and related methods to view *private*

instance variables of objects.

• Send Behavior>>allInstances to query instances of a class.

• The messages class, isKindOf:, respondsTo: etc. are useful for gathering

metrics or building development tools, but they should be avoided in

regular applications: they violate the encapsulation of objects and make

your code harder to understand and maintain.

• SystemNavigation is a utility class holding many useful queries for navigation and browsing the class hierarchy. For example, use SystemNavigation default browseMethodsWithSourceString: 'pharo' matchCase:true. to find and browse all methods with a given source

string. (Slow, but thorough!)

• Every Pharo class points to an instance of MethodDictionary which

maps selectors to instances of CompiledMethod. A compiled method

knows its class, closing the loop.

• RGMethodDefinition is a leightweight proxy for a compiled method,

providing additional convenience methods, and used by many Pharo

tools.

• RBBrowserEnvironment, part of the Refactoring Browser infrastructure,

offers a more refined interface than SystemNavigation for querying the

system, since the result of a query can be used as a the scope of a new

query. Both GUI and programmatic interfaces are available.

• thisContext is a pseudo-variable that reifies the runtime stack of th the

virtual machine. It is mainly used by the debugger to dynamically

construct an interactive view of the stack. It is also especially useful for

dynamically determining the sender of a message.

• Intelligent breakpoints can be set using haltIf:, taking a method selector

as its argument. haltIf: halts only if the named method occurs as a

sender in the run-time stack.

• A common way to intercept messages sent to a given target is to use

a *minimal object* as a proxy for that target. The proxy implements as

few methods as possible, and traps all message sends by implementing

doesNotunderstand:. It can then perform some additional action and

then forward the message to the original target.

• Send become: to swap the references of two objects, such as a proxy and

its target.

Beware, some messages, like class and yourself are never really sent,

but are interpreted by the VM. Others, like +, - and ifTrue: may be

directly interpreted or inlined by the VM depending on the receiver.

• Another typical use for overriding doesNotUnderstand: is to lazily load

or compile missing methods.

• doesNotUnderstand: cannot trap self-sends.

• A more rigorous way to intercept messages is to use an object as a

method wrapper. Such an object is installed in a method dictionary

in place of a compiled method. It should implement run:with:in:

which is sent by the VM when it detects an ordinary object instead of a

compiled method in the method dictionary. This technique is used by

the SUnit Test Runner to collect coverage data.

# Regular expressions

**Syntax What it represents**

a literal match of character a

. match any char (except newline)

(...) group subexpression

\x escape the following special character where ’x’ can be ’w’,’s’,’d’,’W’,’S’,’D’

\* Kleene star — match previous regex zero or more times

+ match previous regex one or more times

? match previous regex zero times or once

| match choice of left and right regex

[abcd] match choice of characters abcd

[^abcd] match negated choice of characters

[0-9] match range of characters 0 to 9

\w match alphanumeric

\W match non-alphanumeric

\d match digit

\D match non-digit

\s match space

\S match non-space

## Examples:

'42' matchesRegex: '\d+'

>>> true

'-1' matchesRegex: '\d+'

>>> false

'0' matchesRegex: '0|([1-9]\d\*)'

>>> true

'1' matchesRegex: '0|([1-9]\d\*)'

>>> true

'42' matchesRegex: '0|([1-9]\d\*)'

>>> true

'099' matchesRegex: '0|([1-9]\d\*)'

>>> false "leading 0"

'0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> true

'-1' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> true

'42' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> true

'+99' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> true

'-0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> false "negative zero"

'01' matchesRegex: '(0|((\+|-)?[1-9]\d\*))'

>>> false "leading zero"

'0' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> true

'0.9' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> true

'3.14' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> true

'-42' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> true

'2.' matchesRegex: '(0|((\+|-)?[1-9]\d\*))(\.\d+)?'

>>> false "need digits after ."

'-999.999e+21' matchesRegex: '(\+|-)?\d+(\.\d\*)?((e|E)(\+|-)?\d+)?'

>>> true

## Character classes can also include the following grep(1)-compatible elements:

**Syntax What it represents**

[:alnum:] any alphanumeric

[:alpha:] any alphabetic character

[:cntrl:] any control character (ascii code below 32)

[:digit:] any decimal digit

[:graph:] any graphical character (ascii code above 32)

[:lower:] any lowercase character

[:print:] any printable character (here, the same as [:graph:])

[:punct:] any punctuation character

[:space:] any whitespace character

[:upper:] any uppercase character

[:xdigit:] any hexadecimal character

These elements are components of the character classes, *i.e.*, they have to be enclosed in an extra set of square brackets to form a valid regular expression.

'42' matchesRegex: '[[:digit:]]+'

>>> true

A sequence of characters between colons is treated as a unary selector which

is supposed to be understood by characters. A character matches such an

expression if it answers true to a message with that selector.

'42' matchesRegex: '[0-9]+'

>>> true

'42' matchesRegex: '\d+'

>>> true

'42' matchesRegex: '[\d]+'

>>> true

'42' matchesRegex: '[[:digit:]]+'

>>> true

'42' matchesRegex: ':isDigit:+'

>>> true

## Matching boundaries

**Syntax What it represents**

CARET match an empty string at the beginning of a line

\$ match an empty string at the end of a line

\b match an empty string at a word boundary

\B match an empty string not at a word boundary

\< match an empty string at the beginning of a word

\> match an empty string at the end of a word

Examples:

'hello world' matchesRegex: '.\*\bw.\*'

>>> true "word boundary before w"

'hello world' matchesRegex: '.\*\bo.\*'

>>> false "no boundary before o"

## matching prefixes and ignoring case

'abacus' matchesRegex: '(a|b)+'

>>> false

'abacus' prefixMatchesRegex: '(a|b)+'

>>> true

'ABBA' matchesRegexIgnoringCase: '(a|b)+'

>>> true

'Abacus' matchesRegexIgnoringCase: '(a|b)+'

>>> false

'Abacus' prefixMatchesRegexIgnoringCase: '(a|b)+'

>>> true

## Enumeration interface

| list |

list := OrderedCollection new.

'Jack meet Jill' regex: '\w+' matchesDo: [:word | list add: word].

list

>>> an OrderedCollection('Jack' 'meet' 'Jill')

'Jack meet Jill' regex: '\w+' matchesCollect: [:word | word size]

>>> an OrderedCollection(4 4 4)

'Jack and Jill went up the hill' allRegexMatches: '\w+'

>>> an OrderedCollection('Jack' 'and' 'Jill' 'went' 'up' 'the' 'hill')

## Replacement and translation

'Krazy hates Ignatz' copyWithRegex: '\<[[:lower:]]+\>'

matchesReplacedWith: 'loves'

>>> 'Krazy loves Ignatz'

'Krazy loves Ignatz' copyWithRegex: '\b[a-z]+\b'

matchesTranslatedUsing: [:each | each asUppercase]

>>> 'Krazy LOVES Ignatz'

## Lower-level interface

When you send the message matchesRegex: to a string, the following happens:

• A fresh instance of RxParser is created, and the regular expression

string is passed to it, yielding the expression’s syntax tree.

• The syntax tree is passed as an initialization parameter to an instance of

RxMatcher. The instance sets up some data structure that will work as a

recognizer for the regular expression described by the tree.

• The original string is passed to the matcher, and the matcher checks for

a match.

You can create a matcher using one of the following methods:

• You can send asRegex or asRegexIgnoringCase to the string.

• You can directly invoke the RxMatcher constructor methods forString:

or forString:ignoreCase: (which is what the convenience methods

above will do).

| octal |

octal := '8r[0-9A-F]+' asRegex.

octal matchesIn: '8r52 = 16r2A'

>>> an OrderedCollection('8r52')

| hex |

hex := '16r[0-9A-F]+' asRegexIgnoringCase.

hex matchesIn: '8r52 = 16r2A'

>>> an OrderedCollection('16r2A')

| hex |

hex := RxMatcher forString: '16r[0-9A-Fa-f]+' ignoreCase: true.

hex matchesIn: '8r52 = 16r2A'

>>> an OrderedCollection('16r2A')

'\w+' asRegex matches: 'Krazy'

>>> true

'\w+' asRegex matchesPrefix: 'Ignatz hates Krazy'

>>> true

'\b[a-z]+\b' asRegex search: 'Ignatz hates Krazy'

>>> true "finds 'hates'"

| number |

number := '\d+' asRegex.

number search: 'Ignatz throws 5 bricks'.

number lastResult

>>> true

matchesStream:, matchesStreamPrefix: and searchStream: are analogous

to the above three messages, but takes streams as their argument.

| ignatz names |

ignatz := ReadStream on: 'Ignatz throws bricks at Krazy'.

names := '\<[A-Z][a-z]+\>' asRegex.

names matchesStreamPrefix: ignatz

>>> true

## Subexpression matches

| items |

items := '((\d+)\s\*(\w+))' asRegex.

items search: 'Ignatz throws 1 brick at Krazy'.

items subexpressionCount

>>> 4

items subexpression: 1

>>> '1 brick' "complete expression"

items subexpression: 2

>>> '1 brick' "top subexpression"

items subexpression: 3

>>> '1' "first leaf subexpression"

items subexpression: 4

>>> 'brick' "second leaf subexpression"

items subBeginning: 3

>>> 14

items subEnd: 3

>>> 15

items subBeginning: 4

>>> 16

items subEnd: 4

>>> 21

| date result |

date :=

'(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+(\d\d?)\s\*,\s\*19(\d\d)'

asRegex.

result := (date matches: 'Aug 6, 1996')

ifTrue: [{ (date subexpression: 4) .

(date subexpression: 2) .

(date subexpression: 3) } ]

ifFalse: ['no match'].

result

>>> #('96' 'Aug' '6')

## Enumeration and Replacement

RxMatcher implements

the following methods for iterating over matches within strings: matchesIn:,

matchesIn:do:, matchesIn:collect:, copy:replacingMatchesWith: and

copy:translatingMatchesUsing:

| seuss aWords |

seuss := 'The cat in the hat is back'.

aWords := '\<([^aeiou]|[a])+\>' asRegex. "match words with 'a' in them"

aWords matchesIn: seuss

>>> an OrderedCollection('cat' 'hat' 'back')

aWords matchesIn: seuss collect: [:each | each asUppercase ]

>>> an OrderedCollection('CAT' 'HAT' 'BACK')

aWords copy: seuss replacingMatchesWith: 'grinch'

>>> 'The grinch in the grinch is grinch'

aWords copy: seuss translatingMatchesUsing: [ :each | each asUppercase ]

>>> 'The CAT in the HAT is BACK'

There are also the following methods for iterating over matches within

streams: matchesOnStream:, matchesOnStream:do:, matchesOnStream:collect:,

copyStream:to:replacingMatchesWith: and copyStream:to:translatingMatchesUsing:.

## Error Handling

RegexSyntaxError is raised if a syntax error is detected while parsing a regex

RegexCompilationError is raised if an error is detected while building a matcher

RegexMatchingError is raised if an error occurs while matching (for example, if a bad selector was specified using ':<selector>:' syntax, or because of the matcher’s internal error).

['+' asRegex] on: RegexError do: [:ex | ^ ex printString ]

>>> 'RegexSyntaxError: nullable closure'

## summary

For simple matching, just send matchesRegex: to a string

When performance matters, send asRegex to the string representing the regex, and reuse the resulting matcher for multiple matches

Subexpression of a matching regex may be easily retrieved to an arbitrary

depth

• A matching regex can also replace or translate subexpressions in a new

copy of the string matched

• An enumeration interface is provided to access all matches of a certain

regular expression

• Regexes work with streams as well as with strings.