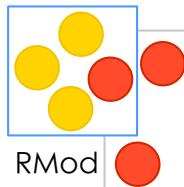


A Little Journey in the Smalltalk Syntax

Stéphane Ducasse
stephane.ducasse@inria.fr
<http://stephane.ducasse.free.fr/>

Goal

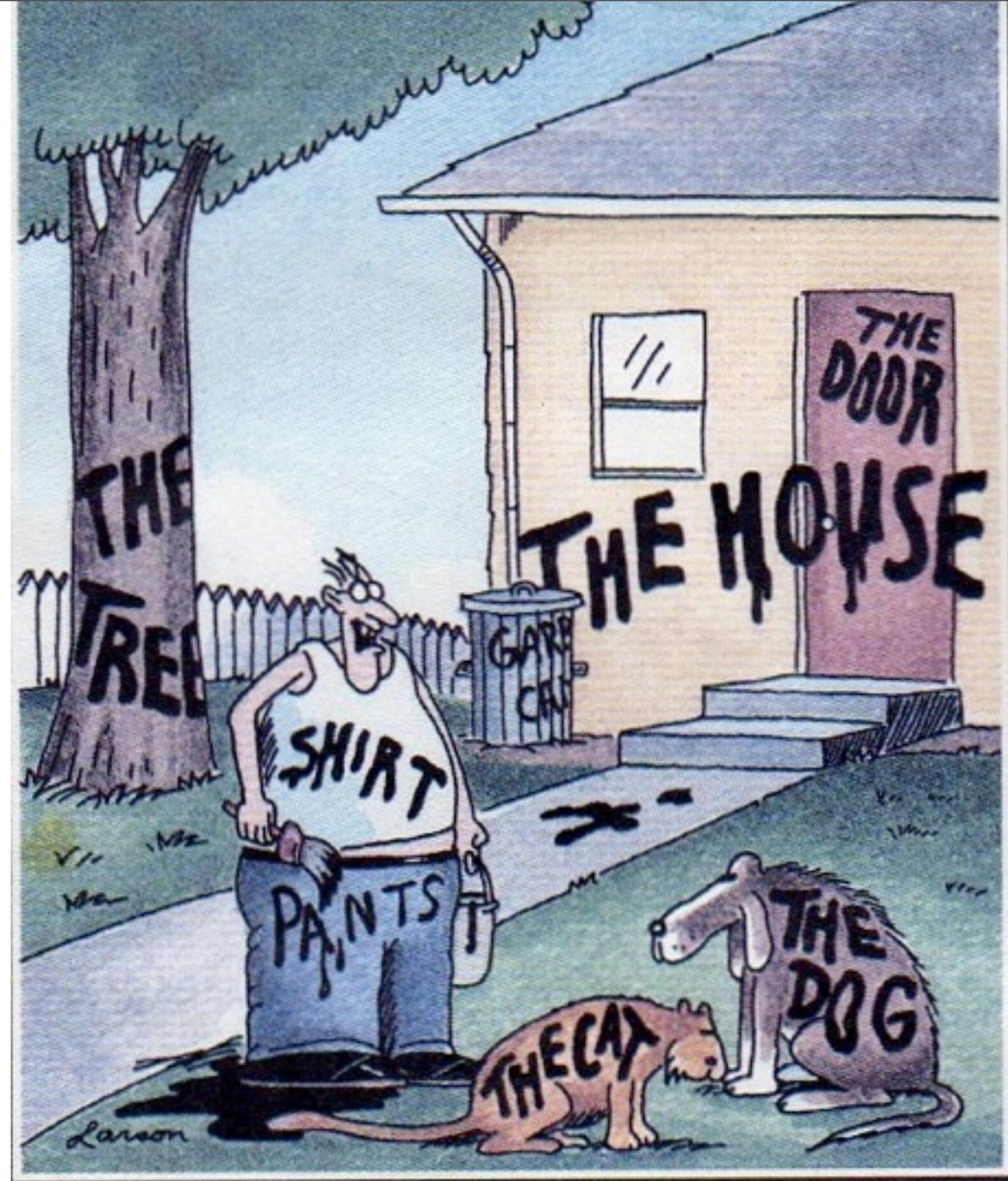


Lower your stress :)
Show you that this is simple

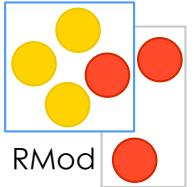


Appetizer!



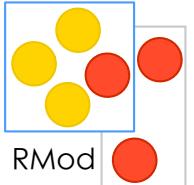


"Now! ... That should clear up
a few things around here!"



Yeah!

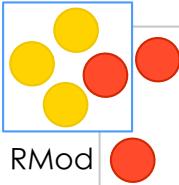
Smalltalk is a dynamically typed language



```
ArrayList<String> strings  
= new ArrayList<String>();
```

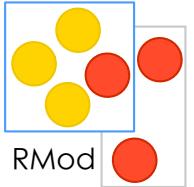
strings := ArrayList new.

Shorter



```
Thread regThread = new Thread(  
    new Runnable() {  
        public void run() {  
            this.doSomething();}  
    });  
regThread.start();
```

[self doSomething] fork.

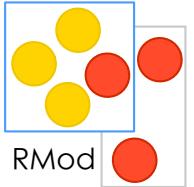


Smalltalk = Objects + Messages + (...)

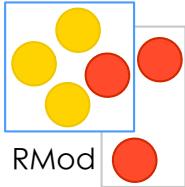
Roadmap

Fun with numbers



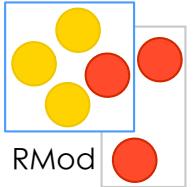


| class

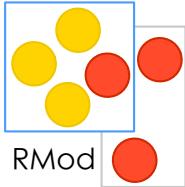


I class

>SmallInteger

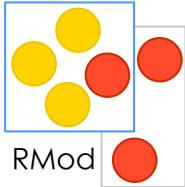


| class maxVal

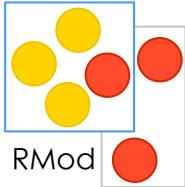


| class maxVal

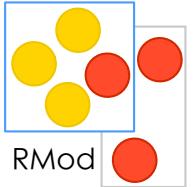
>1073741823



| class maxVal + |



```
| class maxVal + |  
>1073741824
```

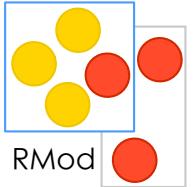


($\lfloor \text{class maxVal} + 1 \rfloor$) class

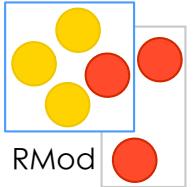
(**I** class maxVal + **I**) class

>LargePositiveInteger

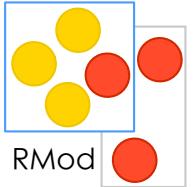




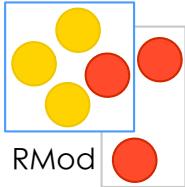
$$(1/3) + (2/3)$$



$$\begin{aligned} & (1/3) + (2/3) \\ & > 1 \end{aligned}$$

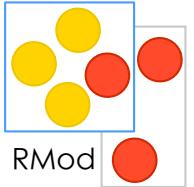


$2/3 + 1$

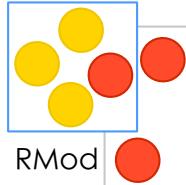


$2/3 + 1$

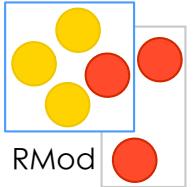
$> 5/3$



1000 factorial



1000 factorial

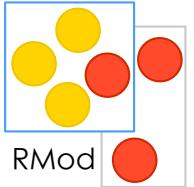


1000 factorial / 999 factorial

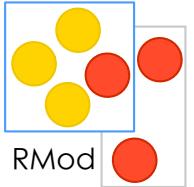
1000 factorial / 999 factorial

> 1000



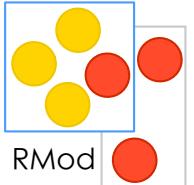


10 @ 100



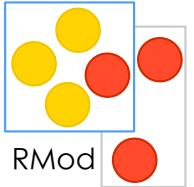
10 @ 100

(10 @ 100) ×



10 @ 100

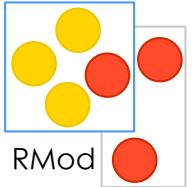
(10 @ 100) ×
≥ 10



10 @ 100

(10 @ 100) x
> 10

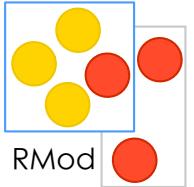
(10 @ 100) y



10 @ 100

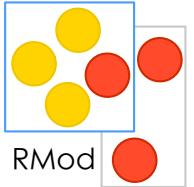
(10 @ 100) x
 > 10

(10 @ 100) y
 > 100



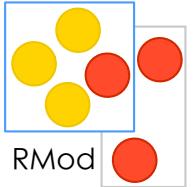
Points!

Points are created using @



Puzzle

(10@100) + (20@100)



Puzzle

$(10@100) + (20@100)$
 $>30@200$

Puzzle

$(10@100) + (20@100)$
 $>30@200$

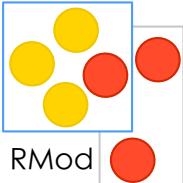


Roadmap

Fun with characters, strings, arrays



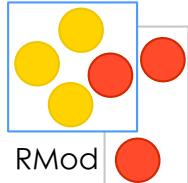
\$C \$h \$a \$r \$a \$c \$t \$e \$r



\$F, \$Q \$U \$E \$N \$T \$i \$N



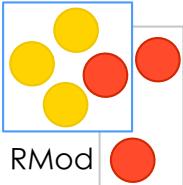
space tab?!



Character space
Character tab
Character cr

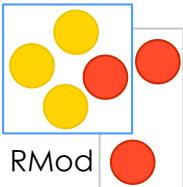


‘Strings’



‘Tiramisu’

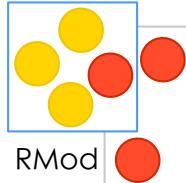
Characters



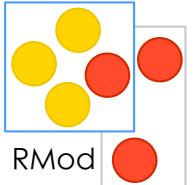
|2 printString

> '|2'

Strings are collections of chars



‘Tiramisu’ at: I



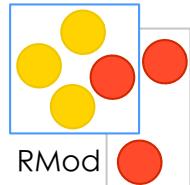
Strings are collections of chars

‘Tiramisu’ at: I

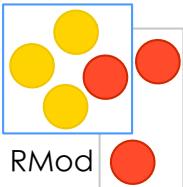
```
> $T
```



A program! -- finding the last char

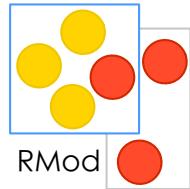


A program!



| str |

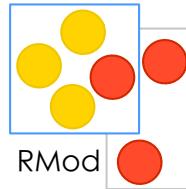
A program!



| str |

local variable

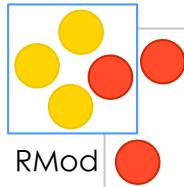
A program!



```
| str |  
str := 'Tiramisu'.
```

local variable

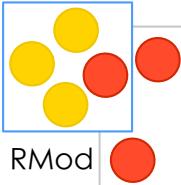
A program!



```
| str |  
str := 'Tiramisu'.
```

local variable
assignment

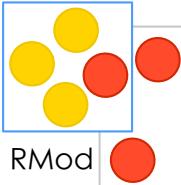
A program!



```
| str |
str := 'Tiramisu'.
str at: str length
```

local variable
assignment

A program!



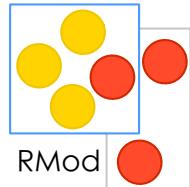
```
| str |
str := 'Tiramisu'.
str at: str length
```

local variable
assignment
message send

```
> $u
```

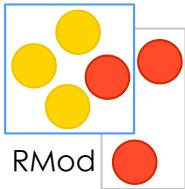


double ' to get one



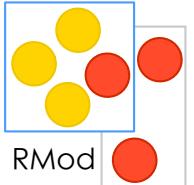
'L'Idiot'

> one string



For concatenation use ,

‘Calvin’ , ‘ & ‘Hobbes’



For concatenation use ,

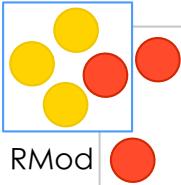
```
'Calvin' , ' & ', 'Hobbes'  
> 'Calvin & Hobbes'
```

For concatenation use ,

‘Calvin’ , ‘ & ‘Hobbes’
 > ‘Calvin & Hobbes’



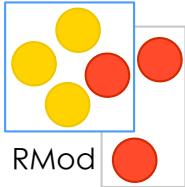
Symbols: #Pharo



`#Something` is a symbol

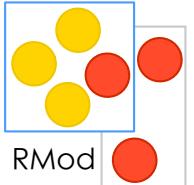
Symbol is a unique string in the system

`#Something == #Something`
> true



“Comment”

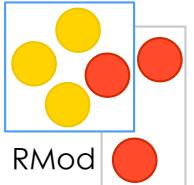
“what a fun language lecture.
I really liked the desserts”



#(Array)

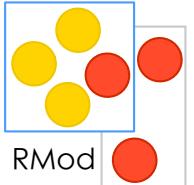
#('Calvin' 'hates' 'Suzie')





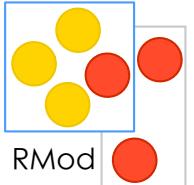
#(Array)

#('Calvin' 'hates' 'Suzie') size



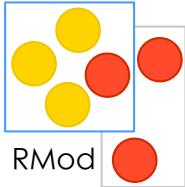
#(Array)

#('Calvin' 'hates' 'Suzie') size
> 3



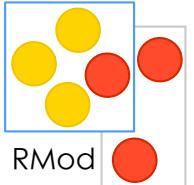
First element starts at 1

`#('Calvin' 'hates' 'Suzie') at: 2`



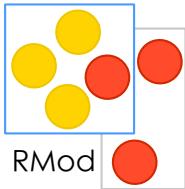
First element starts at 1

```
#('Calvin' 'hates' 'Suzie') at: 2  
> 'hates'
```



at: to access, at:put: to set

```
#('Calvin' 'hates' 'Suzie') at: 2 put:'loves'
```



#(Array)

#('Calvin' 'hates' 'Suzie') at: 2 put:'loves'

> #('Calvin' 'loves' 'Suzie')



Syntax Summary

comment:

“a comment”

character:

\$c \$h \$a \$r \$a \$c \$t \$e \$r \$s \$# \$@

string:

‘a nice string’ ‘lulu’ ‘I’idiot’

symbol:

#mac #+

array:

#(1 2 3 (1 3) \$a 4)

byte array:

#[1 2 3]

integer:

1,2r101

real:

1.5, 6.03e-34, 4, 2.4e7

fraction:

1/33

boolean:

true, false

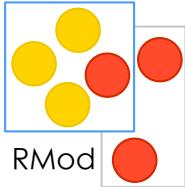
point:

10@120

Roadmap

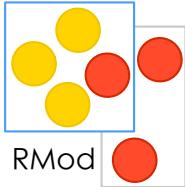
Fun with keywords-based messages





Keyword-based messages

```
arr at: 2 put: 'loves'
```

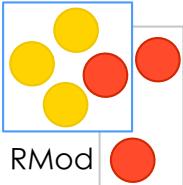


Keyword-based messages

arr at: 2 put: 'loves'

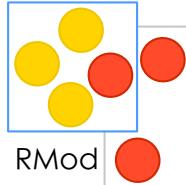
somehow like `arr.atput(2, 'loves')`

From Java to Smalltalk

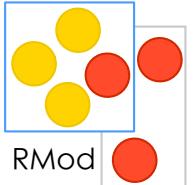


postman.send(mail,recipient);

Removing



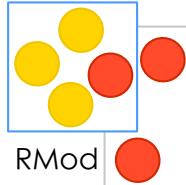
postman.send(mail,recipient);



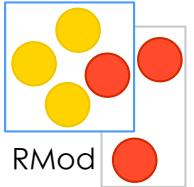
Removing unnecessary

postman send mail recipient

But without losing information



postman send mail **to recipient**



postman send: mail to: recipient

postman.send(mail,recipient);

postman send: mail to: recipient

```
postman.send(mail,recipient);
```

The message is send:to:

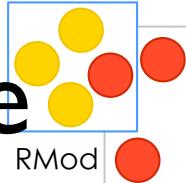


Roadmap

Fun with variables



Shared or Global starts with Uppercase

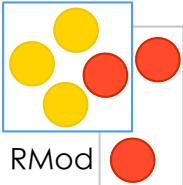


Transcript cr .

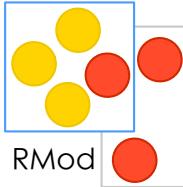
Transcript show:‘hello world’.

Transcript cr .

local or temps starts with lowercase



```
| str |
str := 'Tiramisu'
```



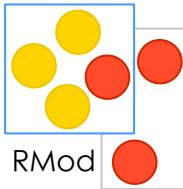
self, super, true, false, nil

self = this

super

true, false are for Booleans

nil is UndefinedObject instance



self, super, true, false, nil

self = this in Java

super

true, false are for Booleans

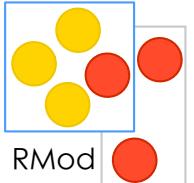
nil is UndefinedObject instance



Roadmap

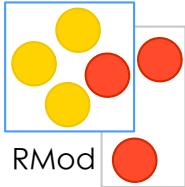
Fun with classes





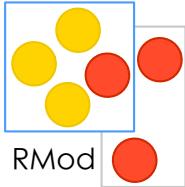
A class definition!

```
Superclass subclass: #Class  
instanceVariableNames: 'a b c'  
...  
category: 'Package name'
```



A class definition!

```
Object subclass: #Point
instanceVariableNames: 'x y'
classVariableNames: ''
poolDictionaries: ''
category: 'Graphics-Primitives'
```



A class definition!

```
Object subclass: #Point
instanceVariableNames: 'x y'
classVariableNames: ''
poolDictionaries: ''
category: 'Graphics-Primitives'
```

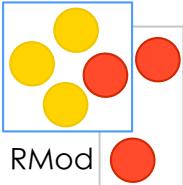


Roadmap

Fun with methods



On Integer

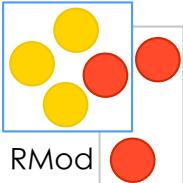


asComplex

"Answer a Complex number that represents value of the the receiver."

^ Complex real: self imaginary: 0

On Boolean

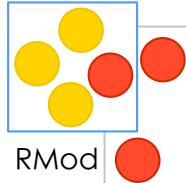


xor: **aBoolean**

"Exclusive OR. Answer true if the receiver is not equivalent to aBoolean."

`^(self == aBoolean) not`

Summary



`self, super`

can access instance variables

can define local variable | ... |

Do not need to define argument types

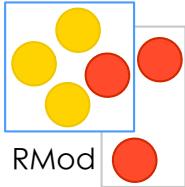
^ to return



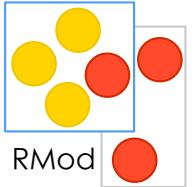
Roadmap

Fun with unary messages

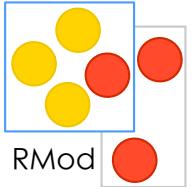




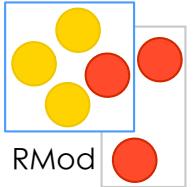
| class



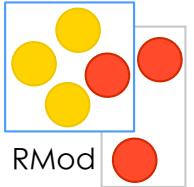
| class
> SmallInteger



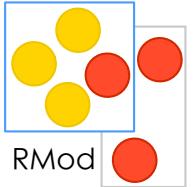
false not



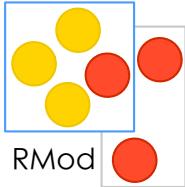
**false not
> true**



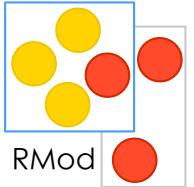
Date today



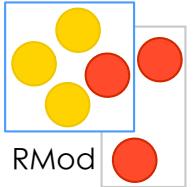
Date today
> 24 May 2009



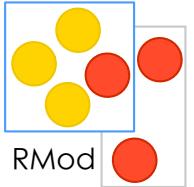
Time now



Time now
> 6:50:13 pm

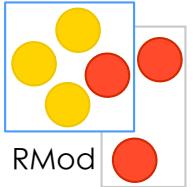


Float pi



Float pi

```
> 3.141592653589793
```



We sent messages to objects or classes!

I class

Date today

We sent messages to objects or classes!

I class
Date today

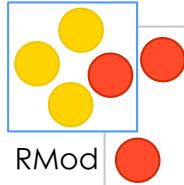


Roadmap

Fun with binary messages



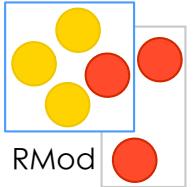
aReceiver aSelector anArgument



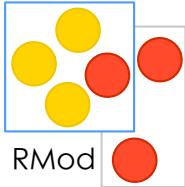
Used for arithmetic, comparison and logical operations

One or two characters taken from:

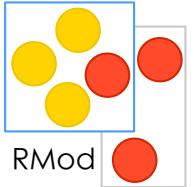
+ - / \ * ~ < > = @ % | & ! ? ,



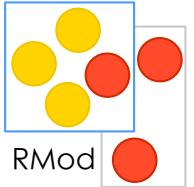
| + 2



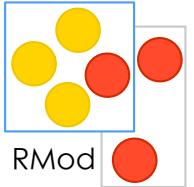
| + 2
>3



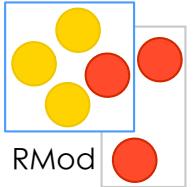
$2 \Rightarrow 3$



```
2 => 3  
> false
```



10 @ 200

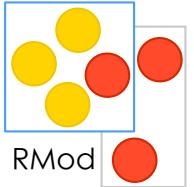


‘Black chocolate’ , ‘ is good’

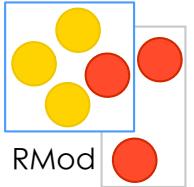
Roadmap

Fun with keyword-based messages

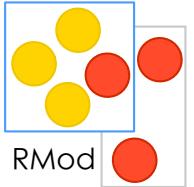




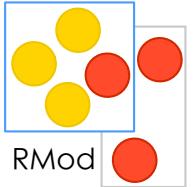
#('Calvin' 'hates' 'Suzie') at: 2 put:'loves'



```
#('Calvin' 'hates' 'Suzie') at: 2 put:'loves'  
> #('Calvin' 'loves' 'Suzie')
```

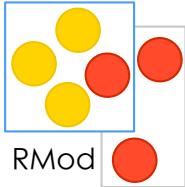


10@20 setX: 2

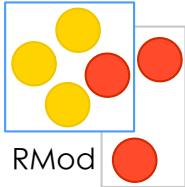


10@20 setX: 2

> 2@20

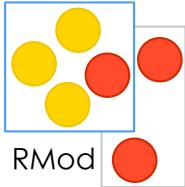


12 between: 10 and: 20



12 between: 10 and: 20

> true



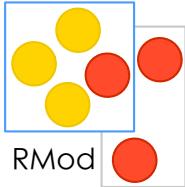
receiver

keyword1: argument1

keyword2: argument2

equivalent to

receiver.keyword1 keyword2(argument1, argument2)



receiver

keyword1: argument1

keyword2: argument2

equivalent to

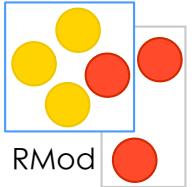
receiver.keyword1 keyword2(argument1, argument2)



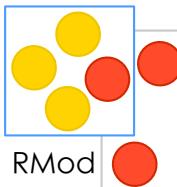
Roadmap

Browser newOnClass: Point





Browser newOnClass: Point



Browser newOnClass: Point

Class Browser: Point

Point

-- all --
*morphic-extent functions
*morphic-truncation and roundoff
*morphicextras-*morphic-postscript canvases
accessing

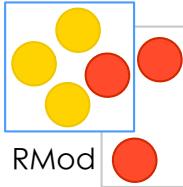
instance ? class

*

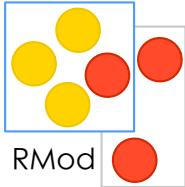
+ - / ..

browse senders implementors versions inheritance hierarchy inst vars class v

Object subclass: #Point
 instanceVariableNames: 'x y'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Graphics-Primitives'



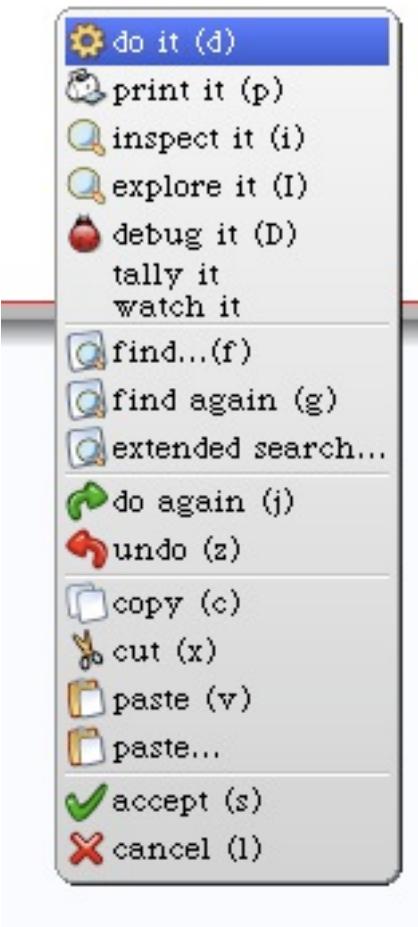
Yes there is a difference between
doing (side effect)
and returning an object



Browser newOnClass: Point
> a Browser

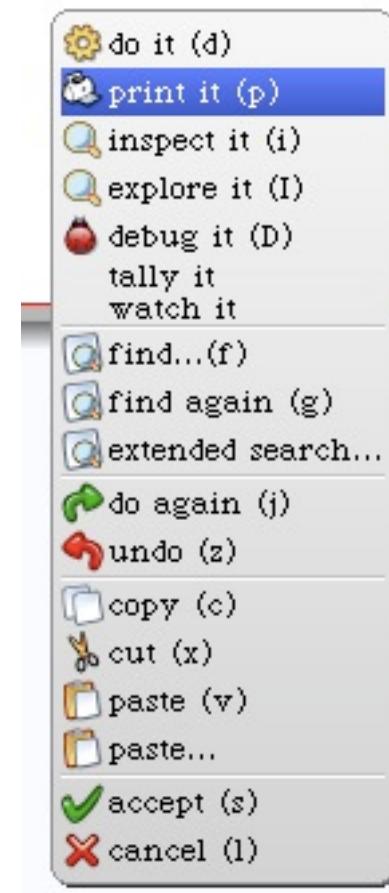
Doing and do not care of the returned result

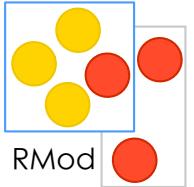
Browser newOnClass: Point



Doing and really want to see the result!

10@20 setX: 2
> 2@20





Doing vs printing (doing + print result)

Doing vs printing (doing + print result)



Roadmap

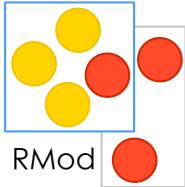
Messages messages

messages

again messages

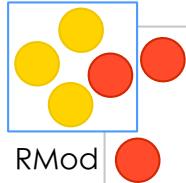
....





Yes there are only messages
unary
binary
keywords

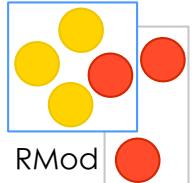
Composition: from left to right!



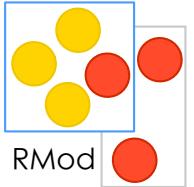
69 class inspect

69 class superclass superclass inspect

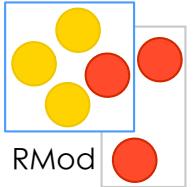
Precedence



Unary> Binary> Keywords

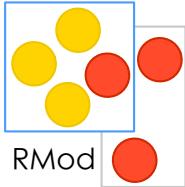


2 + 3 squared



$2 + 3$ squared

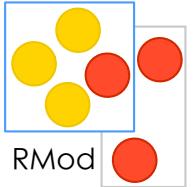
$> 2 + 9$



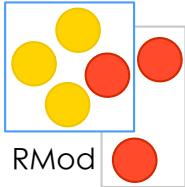
2 + 3 squared

> 2 + 9

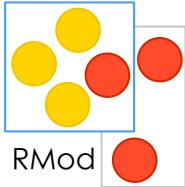
> 11



Color gray - Color white = Color black

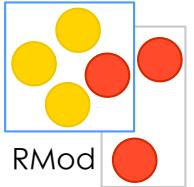


Color gray - Color white = Color black
> aColor = Color black

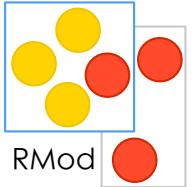


Color gray - Color white = Color black

- > aColor = Color black
- > true

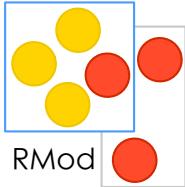


2 raisedTo: 3 + 2



2 raisedTo: 3 + 2

> 2 raisedTo: 5

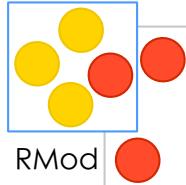


2 raisedTo: 3 + 2

> 2 raisedTo: 5

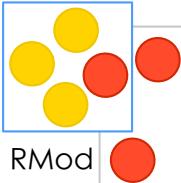
> 32

No mathematical precedence



$1/3 + 2/3$

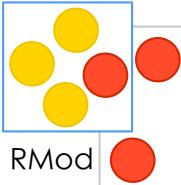
No mathematical precedence



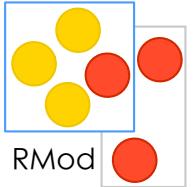
$1/3 + 2/3$

$>7/3 /3$

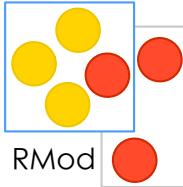
(Msg) > Unary > Binary > Keywords



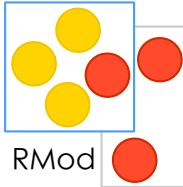
Parenthesized takes precedence!



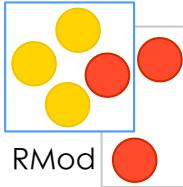
(0@0 extent: 100@100) bottomRight



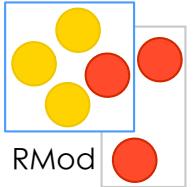
`(0@0 extent: 100@100) bottomRight
> (aPoint extent: anotherPoint)
bottomRight`



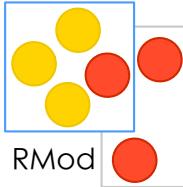
(0@0 extent: 100@100) bottomRight
> (aPoint extent: anotherPoint)
bottomRight
> aRectangle bottomRight



(0@0 extent: 100@100) bottomRight
> (aPoint extent: anotherPoint)
bottomRight
> aRectangle bottomRight
> 100@100



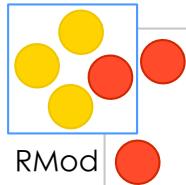
0@0 extent: 100@100 bottomRight



`0@0 extent: 100@100 bottomRight`

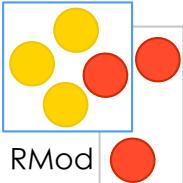
- > Message not understood
- > 100 does not understand bottomRight

No mathematical precedence



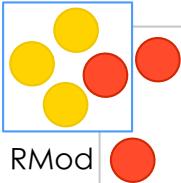
$3 + 2 * 10$

No mathematical precedence



$3 + 2 * 10$
 $> 5 * 10$

No mathematical precedence



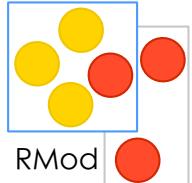
`3 + 2 * 10`

`> 5 * 10`

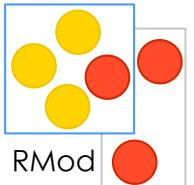
`> 50`

`argh!`

No mathematical precedence



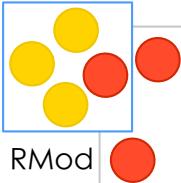
$3 + (2 * 10)$



No mathematical precedence

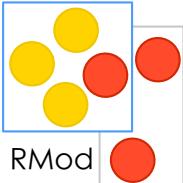
$3 + (2 * 10)$
 $> 3 + 20$

No mathematical precedence



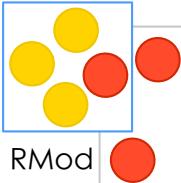
```
3 + (2 * 10)  
> 3 + 20  
> 23
```

No mathematical precedence



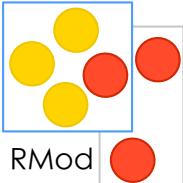
$$\begin{aligned} & 1/3 + 2/3 \\ & > 7/3 / 3 \end{aligned}$$

No mathematical precedence



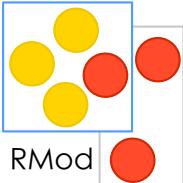
$$\begin{aligned} & 1/3 + 2/3 \\ & > (7/3) / 3 \\ & > 7/9 \end{aligned}$$

No mathematical precedence

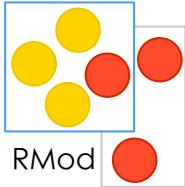


$$(1/3) + (2/3)$$

No mathematical precedence



$$\begin{aligned} & (1/3) + (2/3) \\ & > 1 \end{aligned}$$



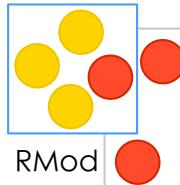
Only Messages

(Msg) > Unary > Binary > Keywords

from left to right

No mathematical precedence

Only Messages



(Msg) > Unary > Binary > Keywords
from left to right
No mathematical precedence

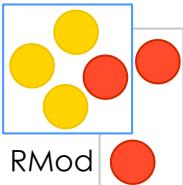


Roadmap

Fun with blocks

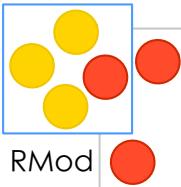


Function definition



$fct(x) = x * x + x$

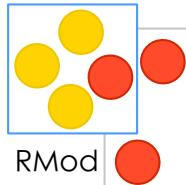
Function Application



`fct (2) = 6`

`fct (20) = 420`

Function definition

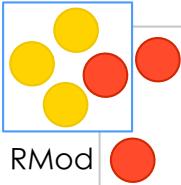


$fct(x) = x * x + x$

|fct|

fct:= [:x | x * x + x].

Function Application



fct (2) = 6

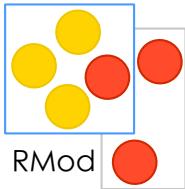
fct (20) = 420

fct value: 2

> 6

fct value: 20

> 420



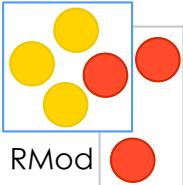
Other examples

[2 + 3 + 4 + 5] value

[:x | x + 3 + 4 + 5] value: 2

[:x :y | x + y + 4 + 5] value: 2 value: 3

Block

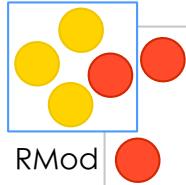


anonymous method

```
[ :variable1 :variable2 |  
  | tmp |  
  expression1.  
  ...variable1 ... ]
```

value: ...

Block



anonymous method

Really really cool!

Can be passed to methods, stored in instance variables

```
[ :variable1 :variable2 |  
  | tmp |  
  expression1.  
  ...variable1 ... ]
```

value: ...

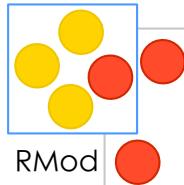


Roadmap

Fun with conditional

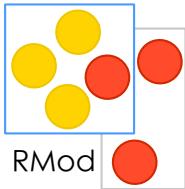


Example



$3 > 0$

```
ifTrue:['positive']
ifFalse:['negative']
```



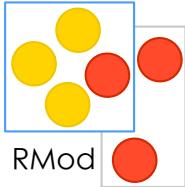
Example

```
3 > 0
```

```
  ifTrue:['positive']
```

```
  ifFalse:['negative']
```

```
> 'positive'
```



Yes **ifTrue:ifFalse:** is a message!

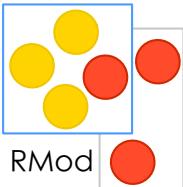
Weather isRaining

ifTrue: [self takeMyUmbrella]

ifFalse: [self takeMySunglasses]

ifTrue:ifFalse is sent to an object: a boolean!

Booleans



& | not

or: and: (lazy)

xor:

ifTrue:ifFalse:

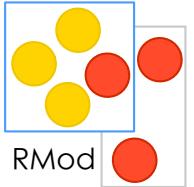
ifFalse:ifTrue:

...

Yes! `ifTrue:ifFalse:` is a message send to a Boolean.

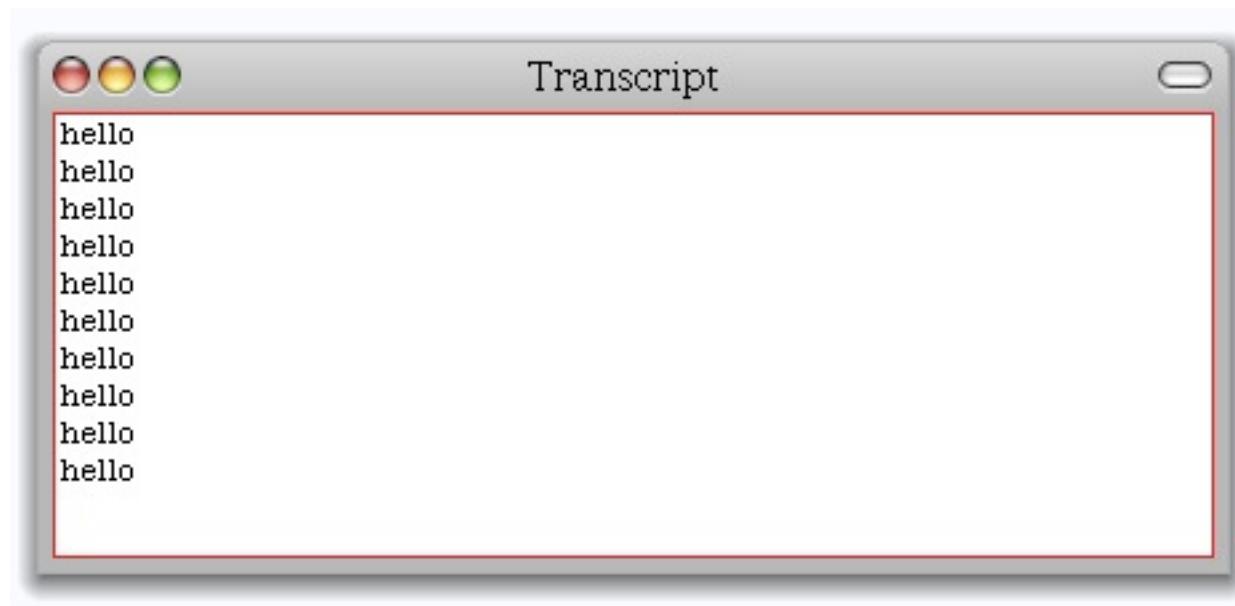
But optimized by the compiler :)

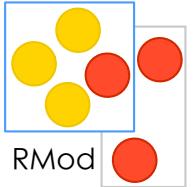




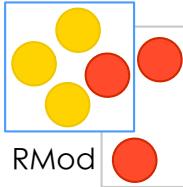
10 timesRepeat: [Transcript show: 'hello'; cr]

10 timesRepeat: [Transcript show: 'hello'; cr]





[$x < y$] whileTrue: [$x := x + 3$]



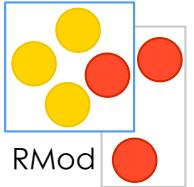
aBlockTest `whileTrue`

aBlockTest `whileFalse`

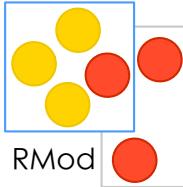
aBlockTest `whileTrue:` *aBlockBody*

aBlockTest `whileFalse:` *aBlockBody*

anInteger `timesRepeat:` *aBlockBody*



Confused with () and [] ?



Only put [] when you do not the number of times something may be executed

(x isBlue) ifTrue: [x schroumph]

10 timesRepeat: [self shout]

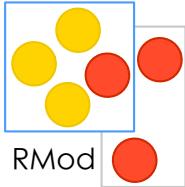
Conditions are messages sent to boolean (x isBlue) ifTrue: []



Roadmap

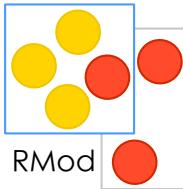
Fun with loops





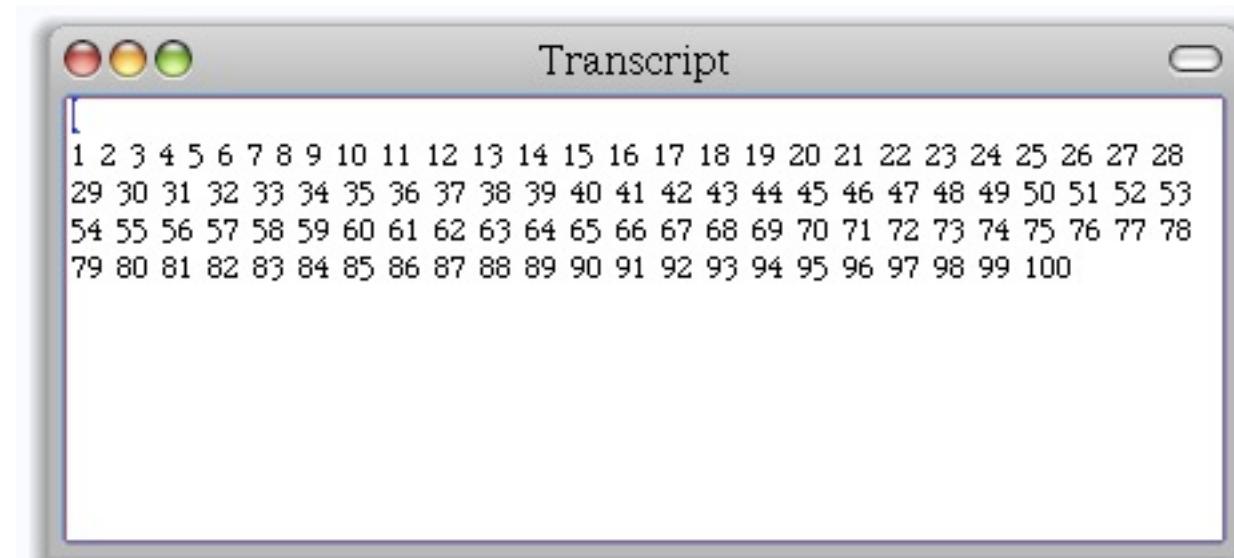
| **to:** | 100 **do:**

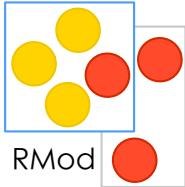
[:i | Transcript show: i ; space]



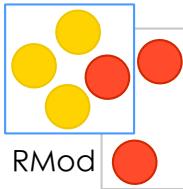
| **to: 100 do:**

[:i | Transcript show: i ; space]

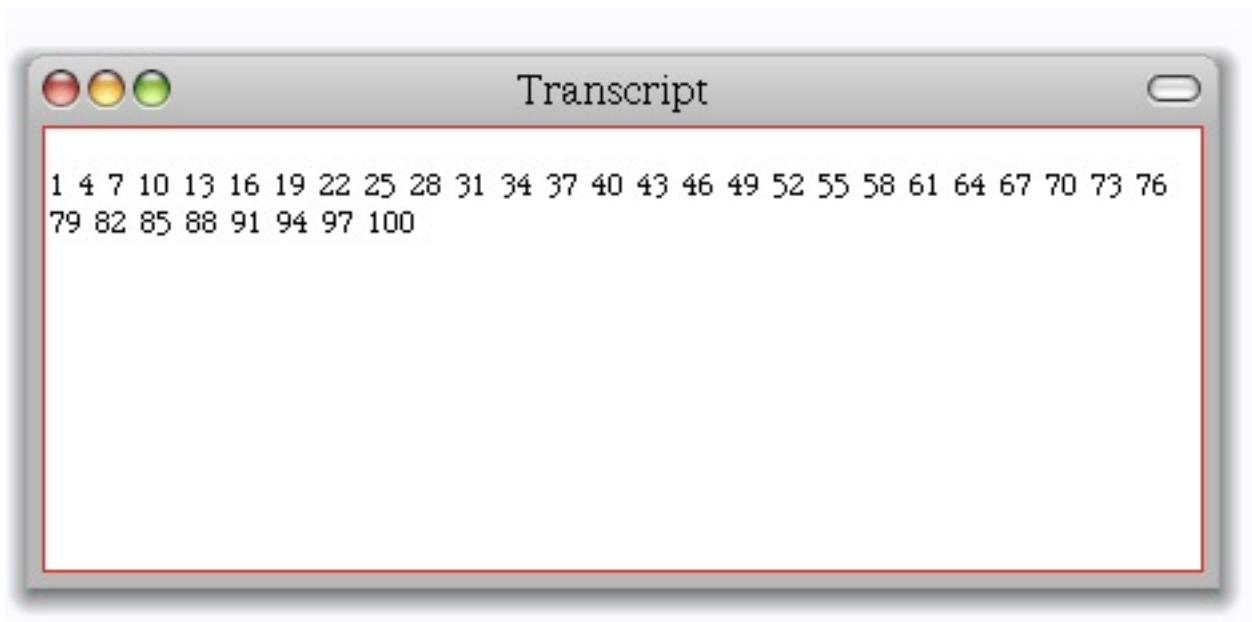


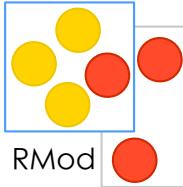


```
| to: 100 by: 3 do:
[ :i | Transcript show: i ; space]
```



```
| to: 100 by: 3 do:  
[ :i | Transcript show: i ; space]
```





So yes there are real loops in Smalltalk!

to:do:

to:by:do:

are just messages send to integers

So yes there are real loops in Smalltalk!

to:do:

to:by:do:

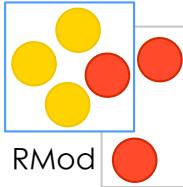
are just messages send to integers



Roadmap

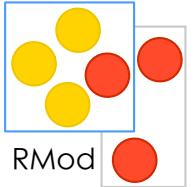
Fun with iterators



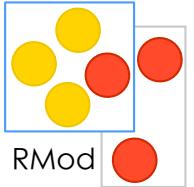


```
ArrayList<String> strings  
    = new ArrayList<String>();  
for(Person person: persons)  
    strings.add(person.name());
```

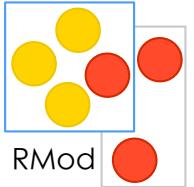
```
strings :=  
persons collect [:person | person name].
```



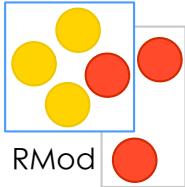
#(2 -3 4 -35 4) collect: [:each| each abs]



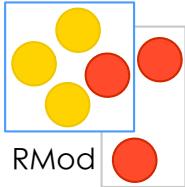
```
#(2 -3 4 -35 4) collect: [ :each| each abs]  
> #(2 3 4 35 4)
```



```
#(15 10 19 68) collect: [:i | i odd ]
```



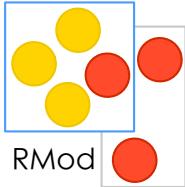
```
#(15 10 19 68) collect: [:i | i odd ]  
> #(true false true false)
```



```
#(15 10 19 68) collect: [:i | i odd ]
```

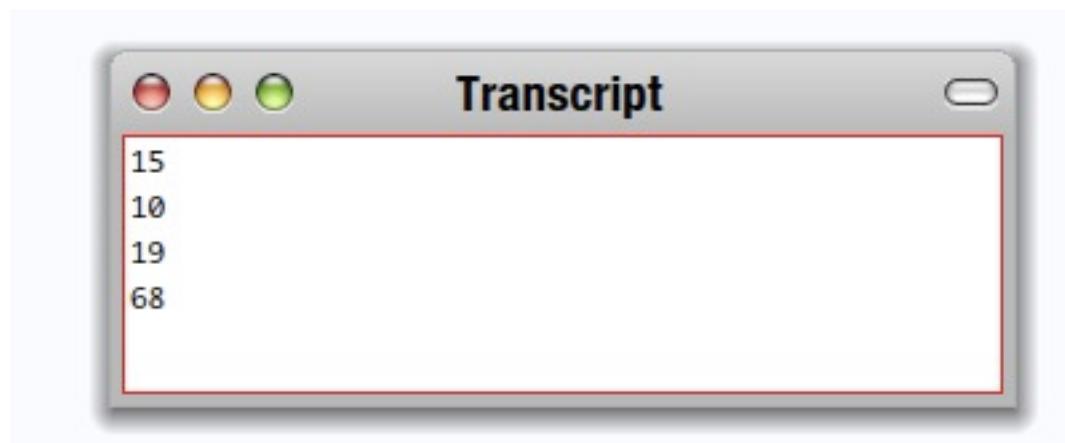
We can also do it that way!

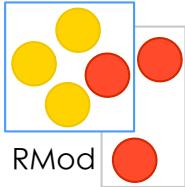
```
|result|
aCol := #( 2 -3 4 -35 4).
result := aCol species new: aCol size.
1 to: aCollection size do:
    [ :each | result at: each put: (aCol at: each) odd].
result
```



```
#(15 10 19 68) do:  
[:i | Transcript show: i ; cr ]
```

```
#(15 10 19 68) do:  
[:i | Transcript show: i ; cr ]
```





```
#(1 2 3)
```

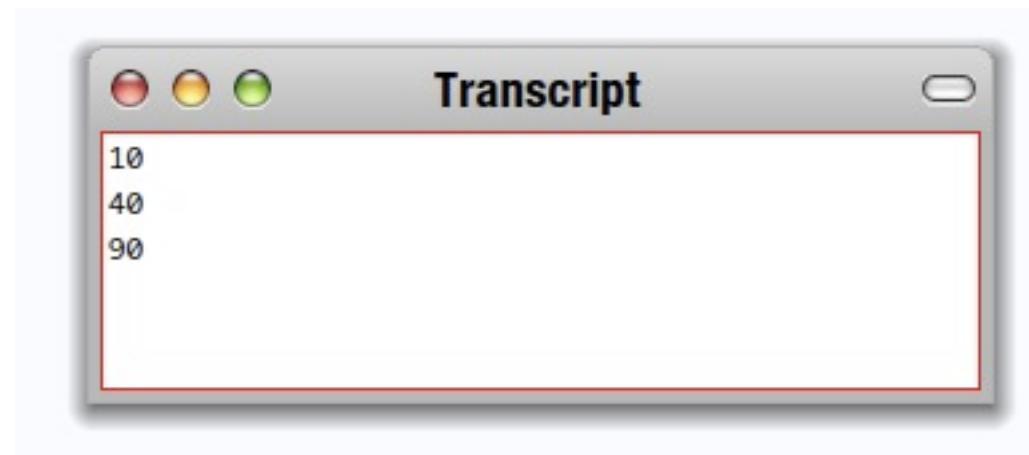
```
with: #(10 20 30)
```

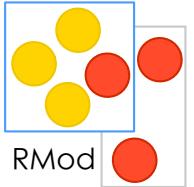
```
do: [:x :y| Transcript show: (y ** x) ; cr ]
```

```
#(1 2 3)
```

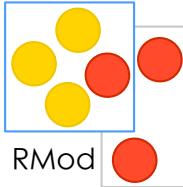
```
with: #(10 20 30)
```

```
do: [:x :y| Transcript show: (y ** x) ; cr ]
```





How do: is implemented?



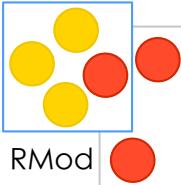
How do: is implemented?

SequenceableCollection>>do: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument."

| **to:** self size **do:** [:i | aBlock value: (self **at:** i)]

Some others... friends

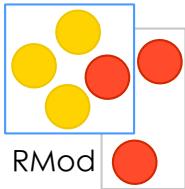


```
#(15 10 19 68) select: [:i|i odd]
```

```
#(15 10 19 68) reject: [:i|i odd]
```

```
#(12 10 19 68 21) detect: [:i|i odd]
```

```
#(12 10 12 68) detect: [:i|i odd] ifNone:[ ]
```



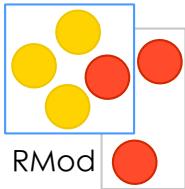
Some others... friends

```
#(15 10 19 68) select: [:i|i odd]  
> #(15 19)
```

```
#(15 10 19 68) reject: [:i|i odd]
```

```
#(12 10 19 68 21) detect: [:i|i odd]
```

```
#(12 10 12 68) detect: [:i|i odd] ifNone:[ ]
```



Some others... friends

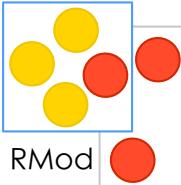
```
#(15 10 19 68) select: [:i|i odd]  
> #(15 19)
```

```
#(15 10 19 68) reject: [:i|i odd]  
> #(10 68)
```

```
#(12 10 19 68 21) detect: [:i|i odd]
```

```
#(12 10 12 68) detect: [:i|i odd] ifNone:[ ]
```

Some others... friends

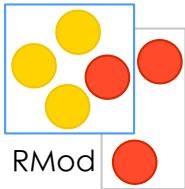


```
#(15 10 19 68) select: [:i|i odd]  
> #(15 19)
```

```
#(15 10 19 68) reject: [:i|i odd]  
> #(10 68)
```

```
#(12 10 19 68 21) detect: [:i|i odd]  
> 19
```

```
#(12 10 12 68) detect: [:i|i odd] ifNone:[ ]
```



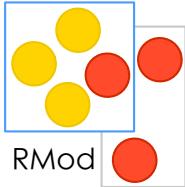
Some others... friends

```
#(15 10 19 68) select: [:i|i odd]  
> #(15 19)
```

```
#(15 10 19 68) reject: [:i|i odd]  
> #(10 68)
```

```
#(12 10 19 68 21) detect: [:i|i odd]  
> 19
```

```
#(12 10 12 68) detect: [:i|i odd] ifNone:[ ]  
> 1
```



Iterators are your best friends

compact

nice abstraction

Just messages sent to collections

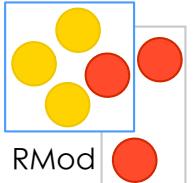
Iterators are your best friends

compact

nice abstraction

Just messages sent to collections

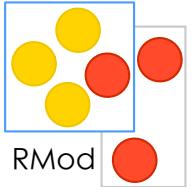




A simple exercise

How do you define the method that does that?

```
#() -> "  
#(a) ->'a'  
#(a b c) -> 'a, b, c'
```



#(a b c)

```
do: [:each | Transcript show: each printString]  
separatedBy: [Transcript show: ',']
```

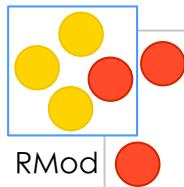
#(a b c)

```
do: [:each | Transcript show: each printString]  
separatedBy: [Transcript show: ',']
```





Messages Sequence



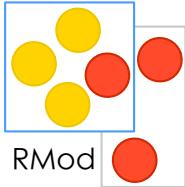
message1 .

message2 .

message3

. is a separator, not a terminator

```
| macNode pcNode node1 printerNode |
macNode := Workstation withName: #mac.
Transcript cr.
Transcript show: 1 printString.
Transcript cr.
Transcript show: 2 printString
```



Multiple messages to an objects ;

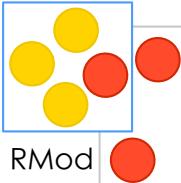
To send multiple messages to the same object

Transcript show: l printString.
Transcript cr

is equivalent to:

Transcript show: l printString ; cr

Hints ...



`x isNil ifTrue: [...]`

`x includes: 3 ifTrue: [...]`

is read as the message `includes:ifTrue:`

`(x includes: 3) ifTrue: [...]`



IT'S SAD HOW SOME PEOPLE
CAN'T HANDLE A LITTLE
VARIETY.



Smalltalk is fun

Pure simple powerful

www.seaside.st

(www.dabbledb.com)

www.pharo-project.org

