Programming Languagages (Langages Evolués)

Roel Wuyts
Smalltalk

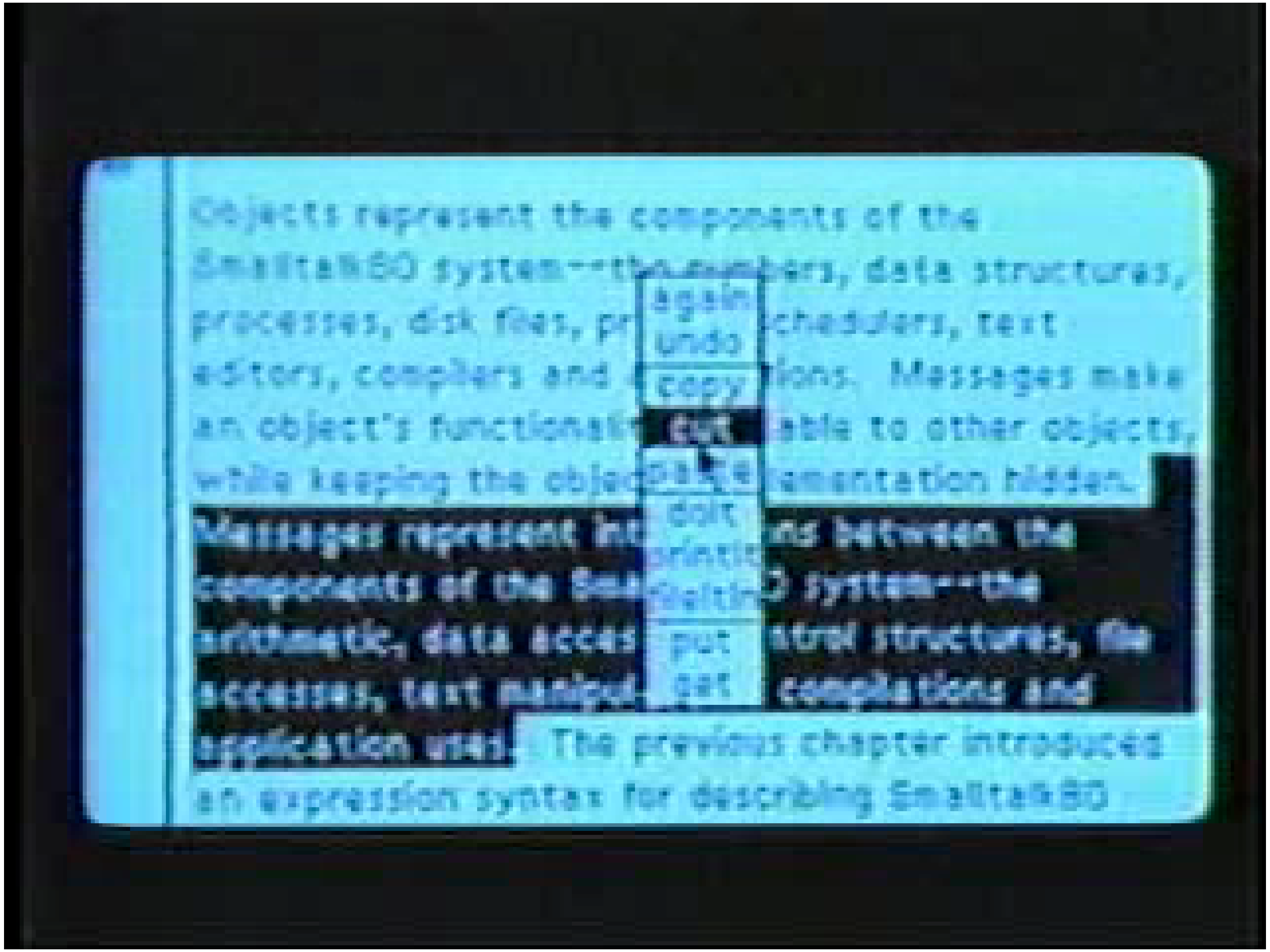
History

- 1962: Simula (Denmark)
  - first object-oriented programming language
  - models the world with objects
  - FYI: C : 1972; C++: 1986
- 1972: Smalltalk '72
- 1980: Smalltalk '80 (standard)
  - all Smalltalk's nowadays are Smalltalk-80
    - but different extensions by vendors...

Smalltalk context

- Xerox-PARC (Palo-alto research center)
- Alan Kay's dynabook (hardware)
- Wanted programming language easy for children
  - Syntax resembles normal sentences
  - Graphical environment
    - First application with multiple, overlapping windows, controlled by a mouse!
    - Bitblt operation

Piece of History...



Objects represent the components of the
Smalltalk80 system--th... ers, data structures,
processes, disk files, pr... chedulers, text
editors, compilers and ... ions. Messages make
an object's functionali... able to other objects,
while keeping the objec... lementation hidden.
Messages represent int... ns between the
components of the Sma... 0 system--the
arithmetic, data acces... ntrol structures, file
accesses, text manipul... compilations and
application uses. The previous chapter introduced
an expression syntax for describing Smalltalk80

Smalltalk at a glance

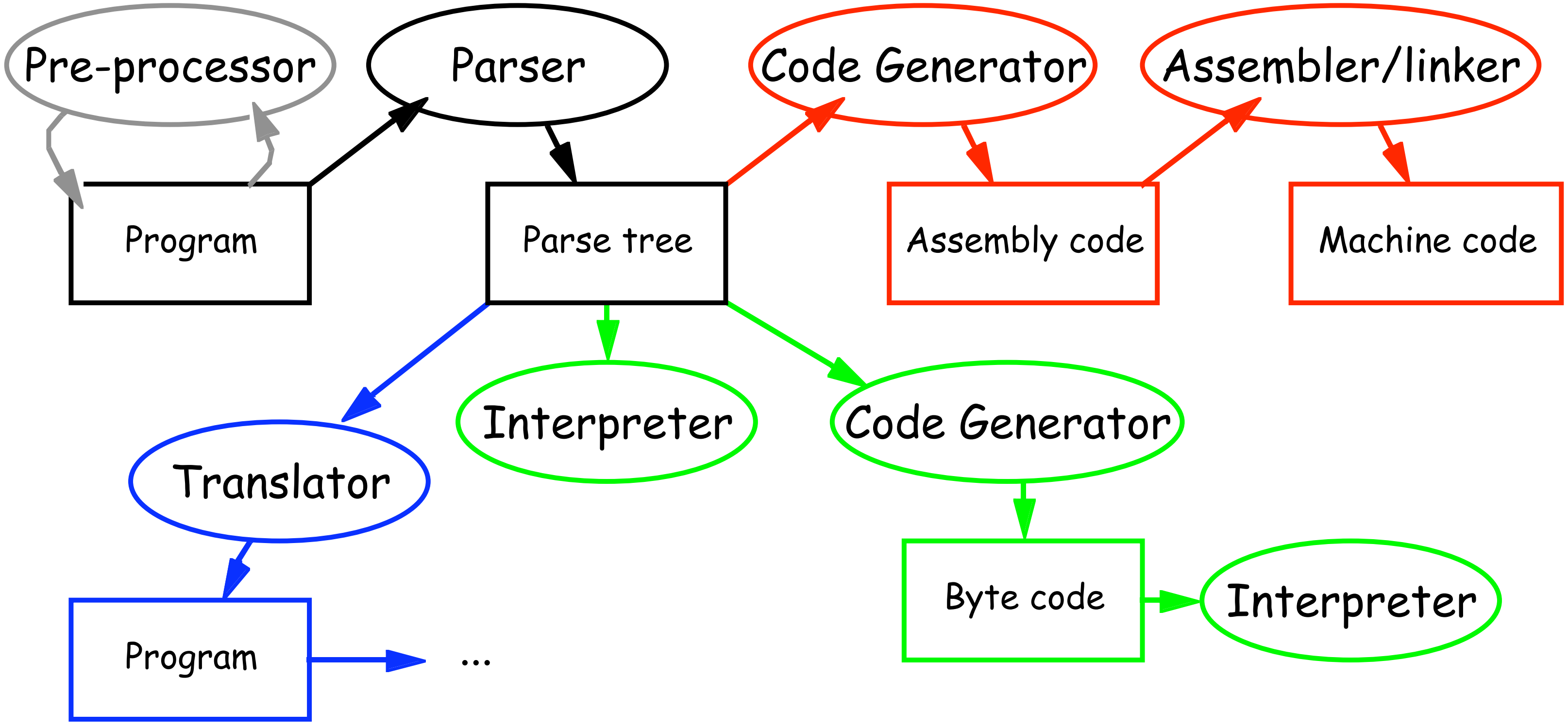
- Pure object-oriented language:
  - *everything* is an object (Integer, Class, Compiler, ...)
  - only message sends (almost no syntax).
  - always late-bound (no statics).
- Meta-programming and (full) reflection
- Dynamically typed
  - no type casts
  - no primitive types

Smalltalk at a glance (ctd)

- Visibility
  - instance variables are private to the object
  - methods are public
- Call by reference (e.g. everything is a pointer)
- Garbage collector
- Single inheritance
- Virtual machine
- Incremental compilation

Concept: Virtual machine

- Compilers and virtual machines have similar front-ends, but different back-ends



Smalltalk syntax

- Three kinds of message sends:
  - unary: ‘Smalltalk course’ printString
  - binary: 1 + 3 or 2@5
  - keyword: 4 > 5 ifTrue: [^’No Way!’] ifFalse: [^’Indeed’]
- Evaluation order: ( ), unary, binary, keyword and left to right
- Pseudo variables:
  - self, super , true, false , nil , thiscontext

Syntax

comment	“a comment”
character	$c $t $e $r $# $@
string	‘a string’ ‘t’’s’
symbol	#mac #+
array	#(1 2 3 (1 3) $a 4)
integer	1, 2r101
real number	1.5, 6.03e-34,4, 2.4e7
fraction	1/33
boolean	true, false

Syntax

assignment	*var := aValue*
block	[ ]
local variable	| tmp1 tmp2 |
block variable	:var
separator	*expr1 . expr2*
return	^ *expr*

Syntax

- Everything else are messages sent to objects!
- Examples

```
(5 > 4) ifTrue: ....
x bitShift: 2
1 to: 10 do: ...
```

- Advantages
  - minimal parsing
    - simple parse tree; ideal for OO research
  - language is extensible

Delayed Evaluation: Blocks

- Blocks function (almost) like block closures
- Code inside a block is not directly evaluated.

```
|array |
array := Array with: 1 with: 2.0 with: 3.
array collect: [:each | (each >= 2)
                                          ifTrue: [’larger’]
                                          ifFalse: [’smaller’]]
```

⇒ array with strings ‘smaller’, ‘larger’, ‘larger’

Why blocks?

- Find the differences:

```
|array |
array := Array with: 1 with: 2.0 with: 3.
array collect: [:each | each + 4]



(define mylist ‘(1  2.0  3))
(map  (lambda (x) (+ x 4)) mylist)
```

#(1 2.0 3)

Evaluating blocks

```
| aBlock |
aBlock := [ Transcript show: ‘I am evaluated’].
aBlock value
```

```
| aBlock |
aBlock := [:first :second | Transcript show: ‘now!’. first + second].
aBlock value: 1 value: 2
```

```
| aBlock |
aBlock := [:one :two :three :four :five | one+two+three+four+five].
aBlock valueWithArguments: (1 to: 5) asArray
```

```
| aBlock |
aBlock := [:val | val > 0
                     ifTrue: [val + (aBlock value: val - 1)]
                     ifFalse: [0]].
aBlock value: 6
```

Core classes

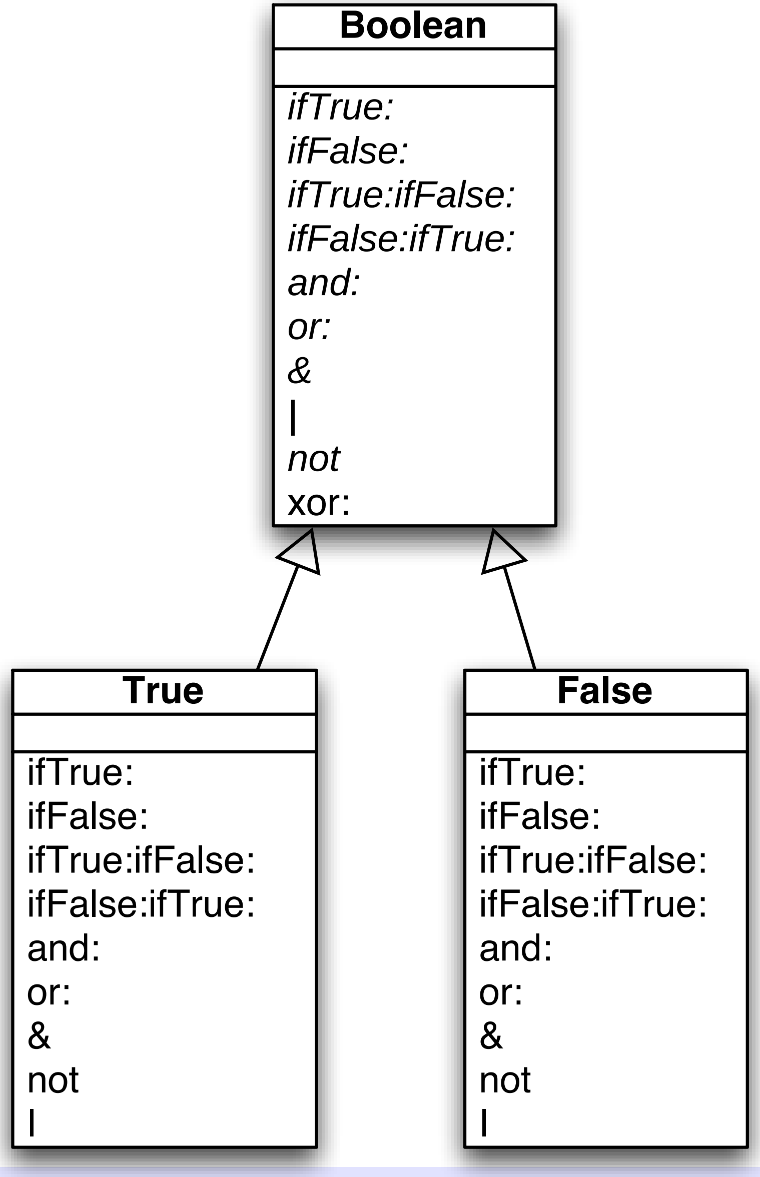
- Let’s have a look at
  - booleans
  - conditionals & loops
  - collections
- All of these are part of the class library
  - not hardcoded in the language!
  - implementation is available in environment
    - learn by example

Booleans

```
2 > 1 ifTrue: [ ... ]

4 < 6 ifFalse: [ ... ]

(Random new next * 10) rounded >= 5
   ifTrue: [Transcript show: ‘Oeh’]
   ifFalse: [Transcript show: ‘Aah’]

4 > 2 & (7 < 9) ifTrue: [ ... ]    “and”

4 > 2 | (9 < 7) ifTrue: [ ... ]    “ or “

(4 < 2 and: [1 / 0 > 8]) ifFalse: [ ... ]    “lazy and”

(4 > 2 or: [1 / 0 > 8]) ifTrue: [ ... ]    “lazy or”

(2 > 4) not ifTrue: [ ... ]
```

Boolean hierarchy

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
   ^falseAlternativeBlock value

ifFalse: alternativeBlock
   ^alternativeBlock value

ifTrue: alternativeBlock
   ^nil

or: alternativeBlock
   ^alternativeBlock value

and: alternativeBlock
   ^self

| aBoolean
   ^aBoolean

& alternativeObject
   ^self

not
   ^true
```



Boolean

ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:
and:
or:
&
|
not
xor:

True

ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:
and:
or:
&
not
|

False

ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:
and:
or:
&
not
|

true and *false*

- true and false are the sole instances of respectively the class True and False
  - Singleton design pattern

Conditionals & Loops

```
| counter max |
max := 10.
number := 1.
[number <= 10] whileTrue: [
   Transcript show: number.
   number := number + 1
]

1 to: 10 do: [:number | Transcript show: number]

1 to 10 by: 3 do: [:i | ...]
```

Conditional & Loop classes

```
BlockClosure>>whileTrue: aBlock

   ^self value
      ifTrue:
         [aBlock value.
         [self value] whileTrue: [aBlock value]]


Number>>to: stop do: aBlock
   (Interval from: self to: stop by: 1) do: aBlock


Number>>to: stop by: step do: aBlock
   (Interval from: self to: stop by: step) do: aBlock
```

Collections

```
| anArray aSet |
anArray := Array with: 1 with: ‘str’ with: Array with: 1.
aSet := aArray asSet.
```

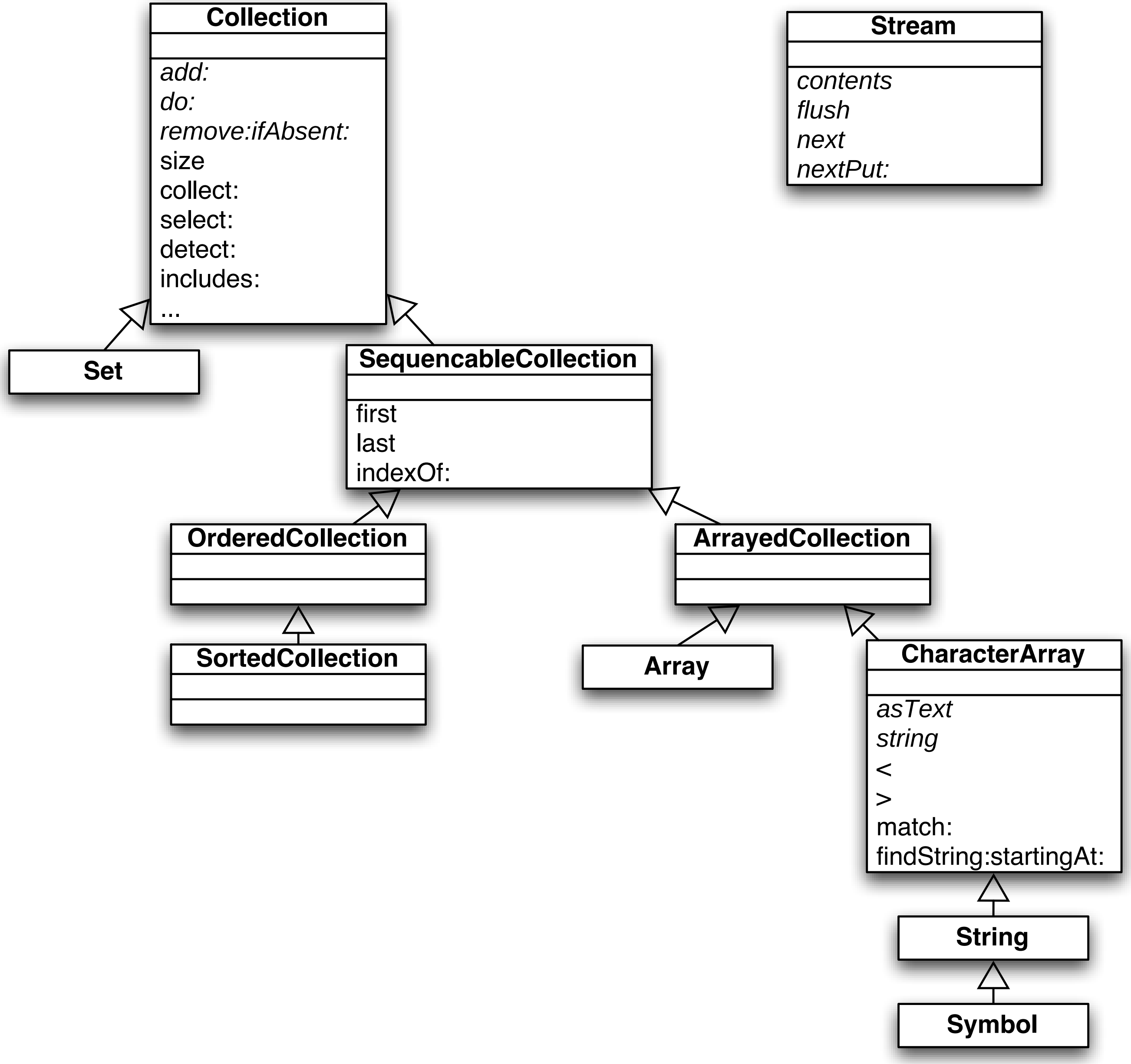
all kinds of objects

```
| dict sortedValues |
dict := Dictionary new.
dict at: $a put: [‘first’].
dict at: $b put: [’little b’].
dict at: $c put: [’another’].
sortedValues := SortedCollection withAll: dict values
                                   sortBlock: [:x :y | x value < y value]
```

```
| weekdays |
weekdays := #(monday tuesday wednesday thursday friday).
weekdays do: [:day | Transcript show: day] separatedBy: [Transcript space]
```

```
| str |
str := ‘mysettings.txt’ asFileName writeStream.
[ str nextPutAll: ‘a string to write’] ensure: [str close]
```

Collection Hierarchy (part)



Example: Infinite streams

- Remember this?

```
(define (integers-from n)
   (cons-stream n (integers-from (+ n 1))))

(define integers (integers-from 1))
```

- We could define list-like operations

```
(filter prime? integers)
```

Dissecting the Scheme streams

takes care of the delaying

Remember stream-car & stream-cdr

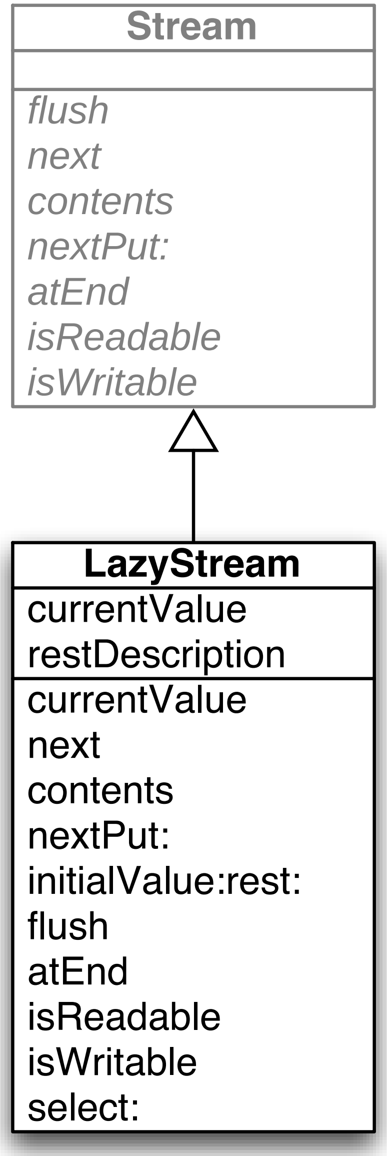
```
(define (integers-from n)
   (cons-stream n (integers-from (+ n 1))))

(define integers (integers-from 1))

(filter prime? integers)
```

initial value

procedure working on streams

Smalltalk Implementation

```
initialValue: anObject rest: description

   currentValue := anObject.
   restDescription := description


currentValue

   ^currentValue


next

   currentValue := restDescription value: self


contents
   | contents |
   contents := OrderedCollection new.
   self next.
   [self atEnd not] whileTrue:
         [contents add: self currentValue.
         self next].
   ^contents
```



Stream

flush
next
contents
nextPut:
atEnd
isReadable
isWritable

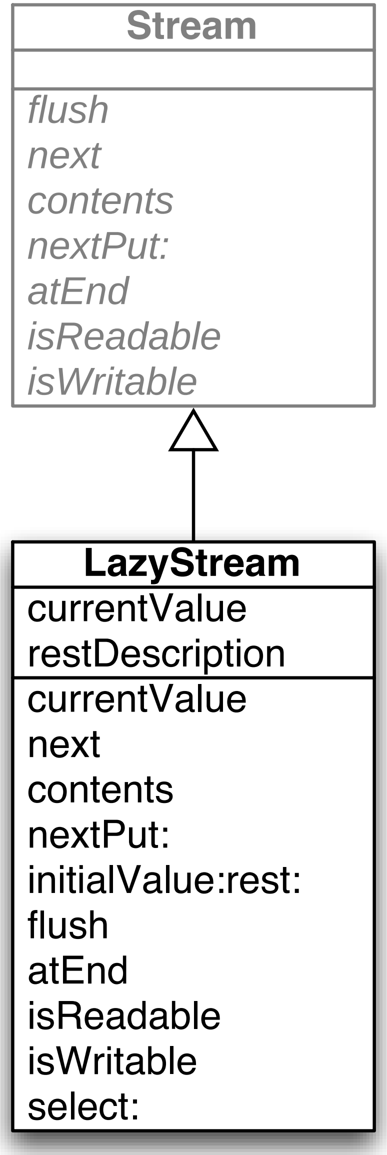
LazyStream

currentValue
restDescription

currentValue
next
contents
nextPut:
initialValue:rest:
flush
atEnd
isReadable
isWritable
select:

Smalltalk implementation (ctd)

```
isReadable
   ^true


isWritable
   ^false


flush
   "do nothing"


nextPut: anObject
   self shouldNotImplement


atEnd
   ^currentValue isNil
```



Smalltalk implementation (ctd)

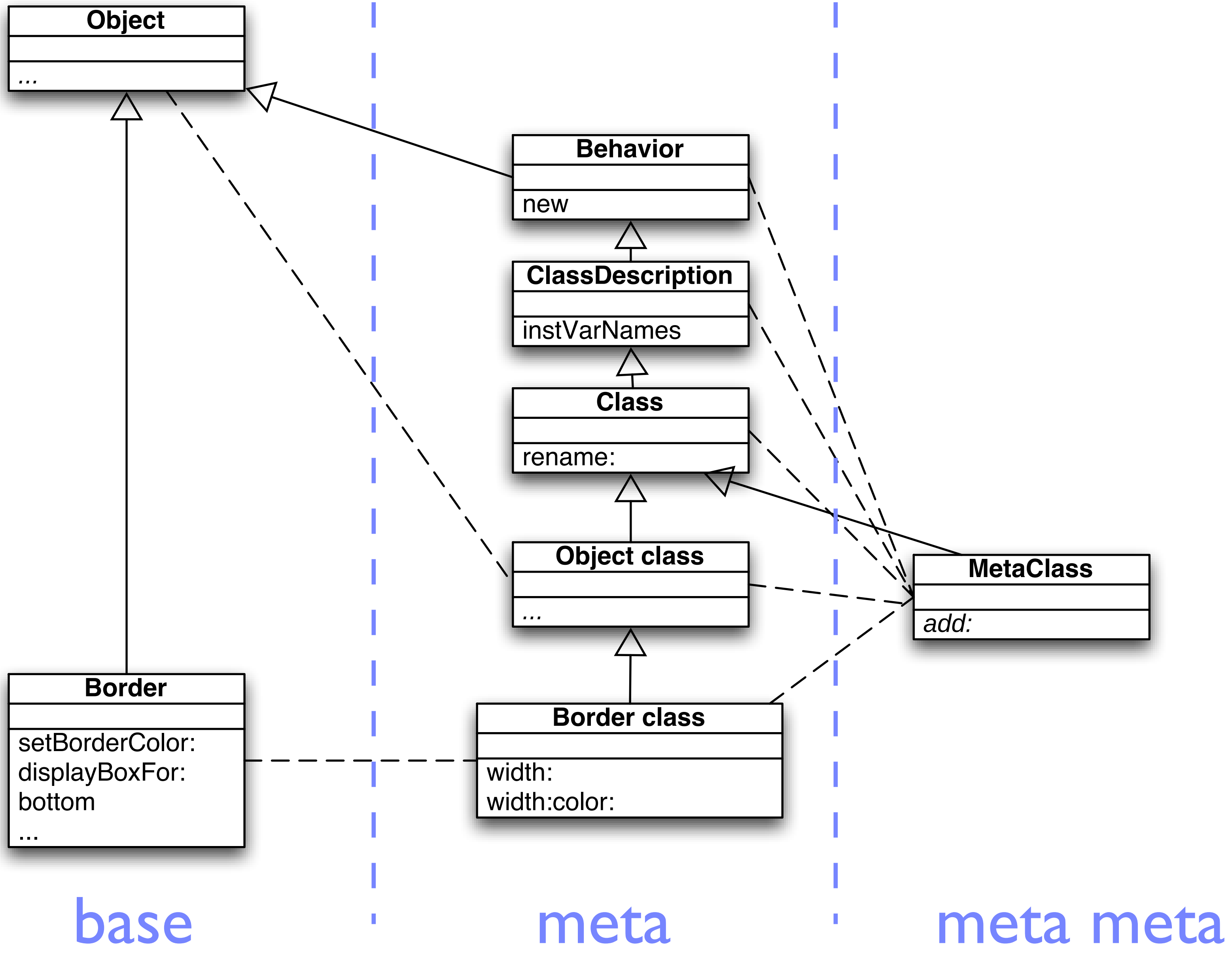
```
select: aBlock

   | str |
   str := LazyStream initialValue: self currentValue rest: [:filteredStr |
      self next.
      [aBlock value: self currentValue] whileFalse: [self next].
      self currentValue].
   (aBlock value: self currentValue) ifFalse: [str next].
   ^str
```

```
"Using the streams"
integersFrom1 := LazyStream
                     initialValue: 1
                     rest: [:str | str currentValue + 1].
evenIntegers := integersFrom1 select: [:each | each even].
evenIntegers currentValue -> 2
evenIntegers next currentValue -> 4
evenIntegers next currentValue -> 6
```

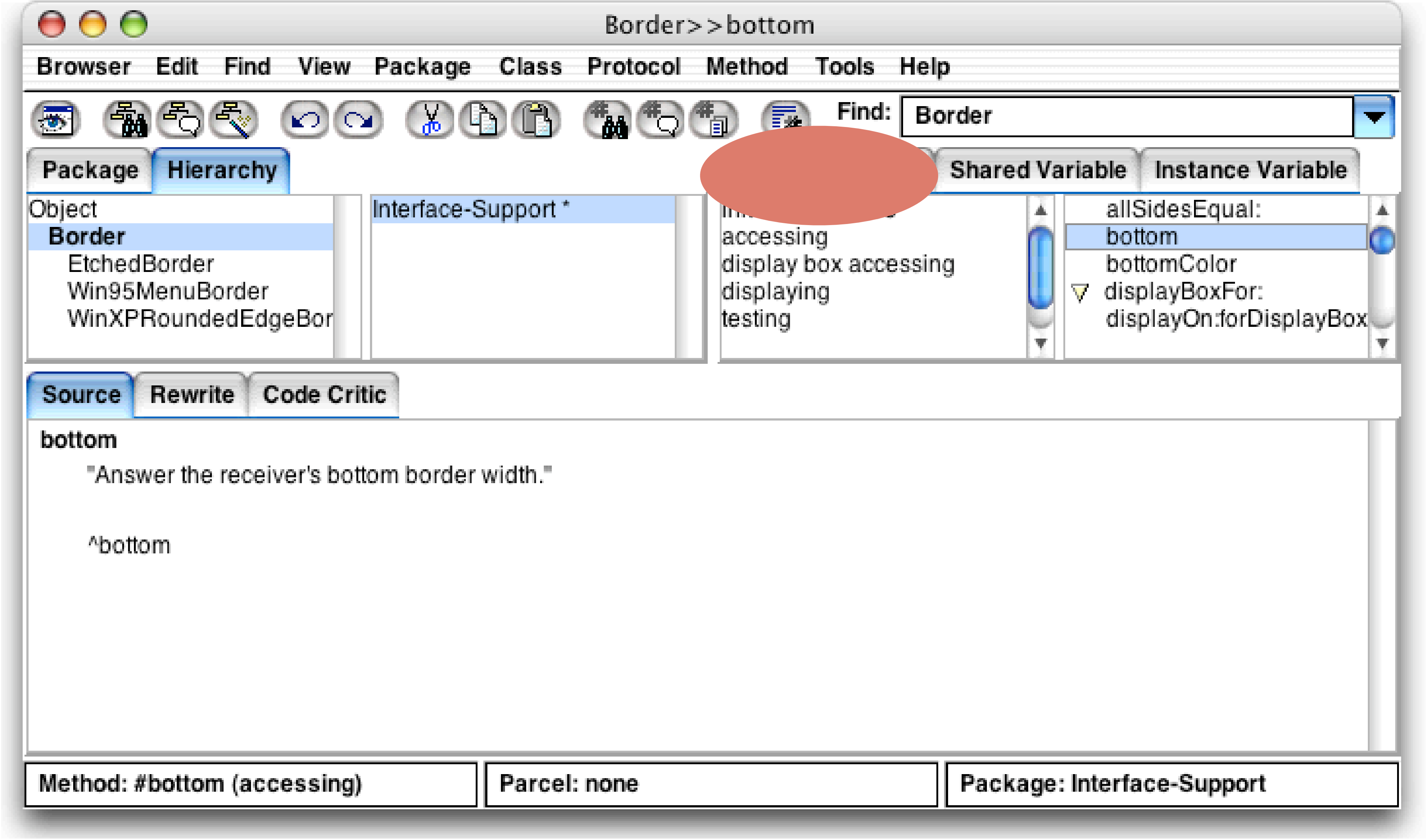
Meta programming in ST

- Everything is an Object
  - Class is an object itself
  - So you can pass it around, store it, compare it, inspect it, send messages to it, ...

Meta system


Object
...
Behavior
new
ClassDescription
instVarNames
Class
rename:
Object class
...
MetaClass
add:
Border
setBorderColor:
displayBoxFor:
bottom
...
Border class
width:
width:color:
base
meta
meta meta

Browser hides complexity



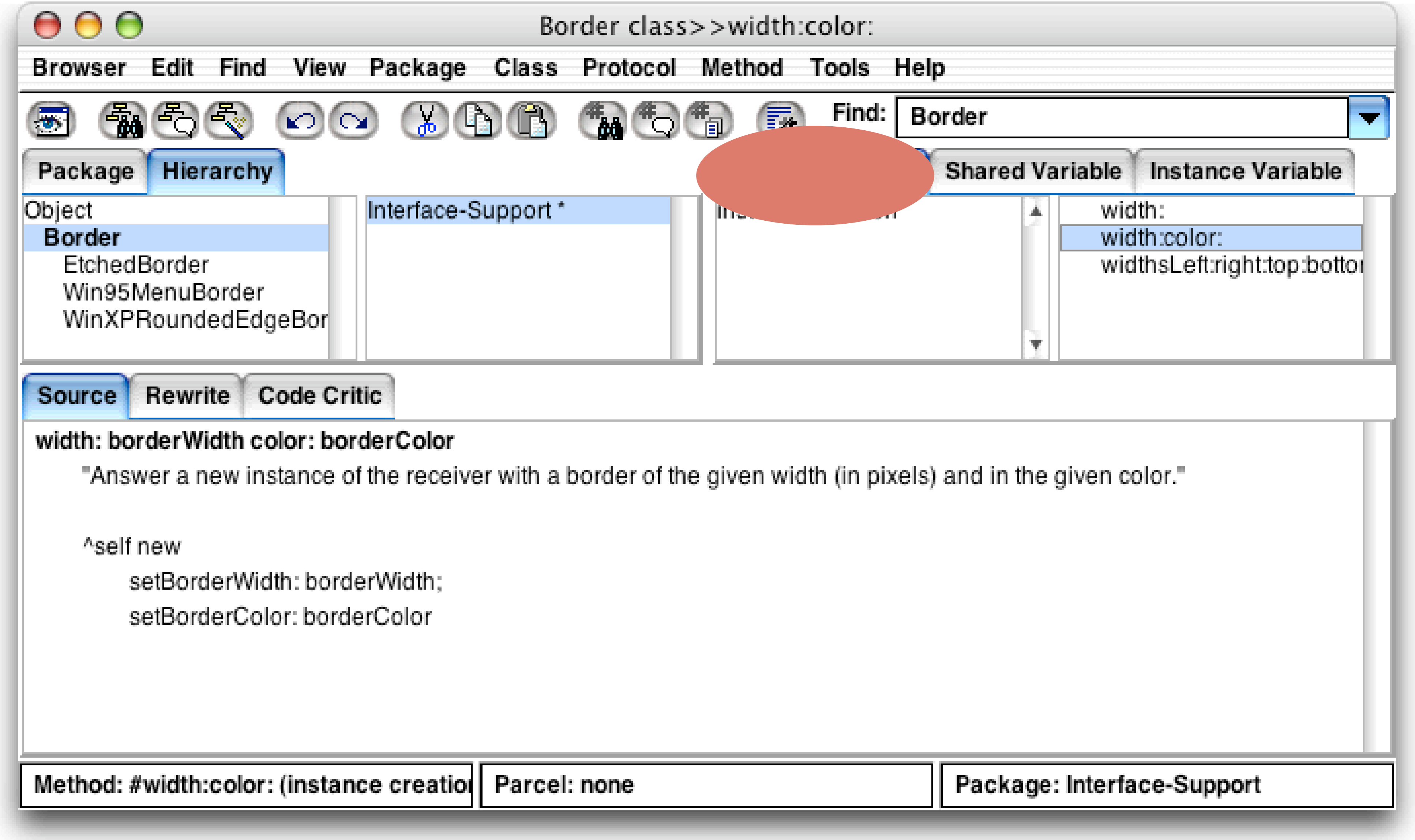
```
bottom
	"Answer the receiver's bottom border width."

	^bottom
```

Method: #bottom (accessing)

Parcel: none

Package: Interface-Support

Browser hides complexity (ctd)



```
width: borderWidth color: borderColor
	"Answer a new instance of the receiver with a border of the given width (in pixels) and in the given color."

	^self new
		setBorderWidth: borderWidth;
		setBorderColor: borderColor
```

Method: #width:color: (instance creatioı

Parcel: none

Package: Interface-Support

No need for constructors

- No special constructors needed
- Just methods
  - Can be inherited, extended, ...
- Example

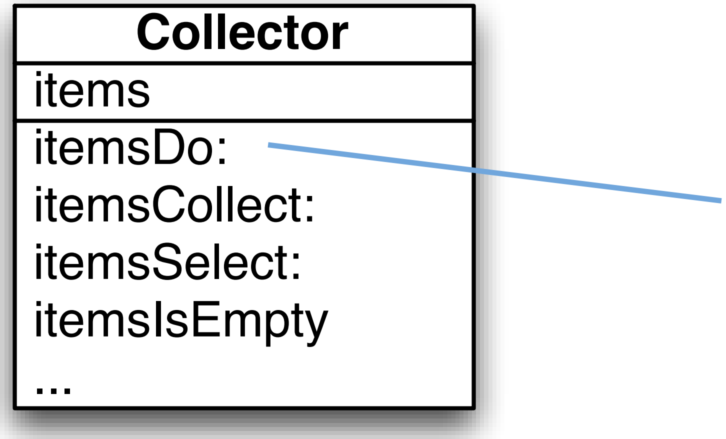
Reflection

- Smalltalk program can, at runtime
  - ask information about itself (introspection)
  - change itself (intercession)
- Examples

```
(2@3) class
(2@3) class class
(2@3) class class class
(2@3) perform: #x
(2@3) perform: #x: arg: 5
(2@3) class selectors
```

Example: Scaffolding Pattern

- We want to have a class that keeps some items in a collection, and that allows to enumerate the elements in that collection



```
itemsDo: aBlock
         ^items do: aBlock
```

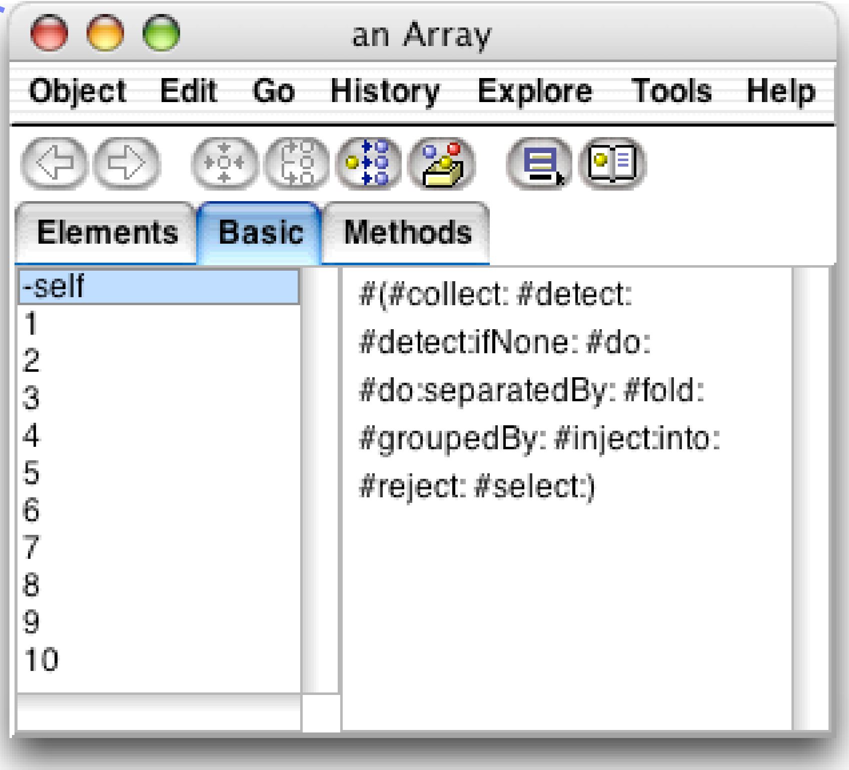
- So we add all these enumeration methods...

Static generation

```
"Let's generate these methods statically"

| enumerationSelectors code codeTemplate |
codeTemplate := '<1s><n><t>"Generated Automatically"<n><n>
<t>^items <1s>'.

enumerationSelectors := Collection organization
                             listAtCategoryNamed: #enumerating.

enumerationSelectors do: [:selector |
   code := WriteStream on: String new.
   selector keywords with: (1 to: selector numArgs)
     do:[:keyword :nr |
          code nextPutAll: keyword; space;
                nextPutAll: arg; print: nr; space].
   Collector
     compile: (codeTemplate expandMacrosWith: code contents)
     classified: #enumerating
]
```

Static generation

```
“Let’s generate these methods statically”
| enumerationSelectors code codeTemplate ctr |
codeTemplate := ‘<1s><n><t>”Generated Automatically”<n><n>
<t>items <1s>’.

enumerationSelectors := Collection organization
                  listAtCategoryNamed: #enumerating.

enumerationSelectors do: [:select
  code := WriteStream on: String
  selector keywords with: (1 to:
    do:[:keyword :nr |
          code nextPutAll: keywor
               nextPutAll: arg;
  Collector
    compile: (codeTemplate expar
    classified: #enumerating
]
```



-self
1
2
3
4
5
6
7
8
9
10

#(#collect: #detect: #detect:ifNone: #do: #do:separatedBy: #fold: #groupedBy: #inject:into: #reject: #select:)

Static generation

```
"Let's generate these methods statically"
| enumerationSelectors code codeTemplate |
codeTemplate := '<1s><n><t>"Generated Automatically"<n><n>
<t>^items <1s>'.

enumerationSelectors := Collection organization
                             listAtCategoryNamed: #enumerating.

enumerationSelectors do: [:selector |
   code := WriteStream on: String new.
   selector keywords with: (1 to: selector numArgs)
     do:[:keyword :nr |
          code nextPutAll: keyword; space;
               nextPutAll: arg; print: nr; space].
   Collector
     compile: (codeTemplate expandMacrosWith: code contents)
     classified: #enumerating
]
```

#inject: 1 -> inject: arg1
#into: 2 -> into: arg2

Static generation

```
"Let's generate these methods statically"
| enumerationSelectors code codeTemplate |
codeTemplate := '<1s><n><t>"Generated Automatically"<n><n>
<t>^items <1s>'.

enumerationSelectors := Collecti
                             lis

enumerationSelectors do: [:selec
   code := WriteStream on: String new.
   selector keywords with: (1 to: selector numArgs)
      do:[:keyword :nr |
            code nextPutAll: keyword; space;
                  nextPutAll: arg; print: nr; space].
   Collector
      compile: (codeTemplate expandMacrosWith: code contents)
      classified: #enumerating
]
```

```
inject: arg1  into: arg2
      "Generated Automatically"

      ^items inject: arg1  into: arg2
```

Let’s forward them to *items*

```
doesNotUnderstand: aMessage

   | enumerationSelectors |
   enumerationSelectors := Collection organization
                               listAtCategoryNamed: #enumerating.

   ^(enumerationSelectors includes: aMessage selector)
      ifTrue: [items
                  perform: aMessage selector
                  withArguments: aMessage arguments
      ifFalse: [super doesNotUnderstand: aMessage]
```

Let’s generate on the fly

```
doesNotUnderstand: aMessage
   | selector |
   selector := aMessage selector.
   (self isEnumerationSelector: selector)
            ifFalse: [^super doesNotUnderstand: aMessage].
   self compileEnumerationMethodFor: selector.
   ^self perform: selector withArguments: aMessage arguments

isEnumerationSelector: selector
   | enumerationSelectors |
   enumerationSelectors := Collection organization
                                    listAtCategoryNamed: #enumerating.
   ^enumerationSelectors includes: selector

compileEnumerationMethodFor: selector
   | codeTemplate code |
   codeTemplate := self enumerationTemplate.
   code := WriteStream on: String new.
   selector keywords with: (1 to: selector numArgs)
      do: [:keyword :nr | code nextPutAll: keyword;
                            space; nextPutAll: ‘arg’; print: nr; space].
   self class
      compile: (codeTemplate expandMacrosWith: code contents)
      classified: #enumerating
```

Wrap-up

- Smalltalk: class-based object-oriented language
- Pure: everything is an object, only message sending
  - simple syntax
  - easy to extend and play with
- Meta-programming & Reflection

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/