INRIA

Smalltalk in a Nutshell

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

Goals

Syntax in a Nutshell
OO Model in a Nutshell

Smalltalk OO Model

Everything is an object
  Only message passing
  Only late binding
Instance variables are private to the object
Methods are public
Everything is a pointer

Garbage collector
Single inheritance between classes
Only message passing between objects

Complete Syntax on a PostCard

```
exampleWithNumber: x
"A method that illustrates every part of Smalltalk method syntax except primitives. It
has unary, binary, and key word messages, declares arguments and temporaries (but
not block temporaries), accesses a global variable (but not and instance variable), uses
literals (array, character, symbol, string, integer, float), uses the pseudo variable true
false, nil, self, and super, and has sequence, assignment, return and cascade. It has both
zero argument and one argument blocks. It doesn't do anything useful, though"
    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a 'a' 1 1.0)
        do: [:each | Transcript
                show: (each class name);
                show: (each printString);
                                        show:' '].
    ^ x < y
```


Language Constructs

^	return
"	comments
#	symbol or array
'	string
[ ]	block or byte array
.	separator and not terminator (or namespace access in VW)
;	cascade (sending several messages to the same instance)
|	local or block variable
:=	assignment
$	character
:	end of selector name
e, r	number exponent or radix
!	file element separator
<primitive: ...>	for VM primitive calls

Syntax

comment:	"a comment"
character:	$c $h $a $r $a $c $t $e $r $s $# $@
string:	'a nice string' 'lulu' 'l''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
float:	1/33
boolean:	true, false
point:	10@120

Note that @ is not an element of the syntax, but just a message sent to a number. This is the same for /, bitShift, ifTrue:, do: ...

Syntax in a Nutshell (II)

assigment: var := aValue
block: [:var ||tmp| expr...]

temporary variable:	|tmp|
block variable:	:var
unary message:	receiver selector
binary message:	receiver selector argument
keyword based:	receiver keyword1: arg1 keyword2: arg2...
cascade:	message ; selector ...
separator:	message . message
result:	^
parenthesis:	(...)

Class Definition in St-80

NameOfSuperclass subclass: #***NameOfClass***
  instanceVariableNames: '***instVarName1***'
  classVariableNames: '***classVarName1***'
  poolDictionaries: ''
  category: 'LAN'

Method Definition

- Normally defined in a browser or (by directly invoking the compiler)
- Methods are **public**
- **Always return self**

Node>>accept: thePacket
  "If the packet is addressed to me, print it.
  Else just behave like a normal node"

  (thePacket isAddressedTo: self)
    ifTrue: [self print: thePacket]
    ifFalse: [super accept: thePacket]

Instance Creation: Messages Too!

- '1', 'abc'
- Basic class creation messages are
  new, new:,
  basicNew, basicNew:
  Monster new
- Class specific message creation (messages sent to classes)
  Tomagoshi withHunger: 10

Messages and their Composition

Three kinds of messages
Unary: Node new
Binary: 1 + 2, 3@4
Keywords: aTomagoshi eat: #cooky furiously: true
Message Priority
(Msg) > unary > binary > keywords
Same Level from left to right
Example:
(10@0 extent: 10@100) bottomRight
s isNil ifTrue: [self halt]

Blocks

- Anonymous method
- Passed as method argument or stored
- Functions

fct(x)= x*x+3, fct(2).
fct :=[:x| x * x + 3]. fct value: 2

```
Integer>>factorial
    | tmp |
        tmp:= 1.
        2 to: self do: [:i| tmp := tmp * i]

    #(1 2 3) do: [:each | Transcript show: each printString ; cr]
```


Yes ifTrue: is sent to a boolean

Weather isRaining
ifTrue: [self takeMyUmbrella]
ifFalse: [self takeMySunglasses]

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

Yes a collection is iterating on itself

```
#(1 2 -4 -86)
      do: [:each | Transcript show: each abs
printString ;cr ]
```

> 1
> 2
> 4
> 86

Yes we ask the collection object to

Summary

Objects and Messages
Three kinds of messages
unary
binary
keywords
Block: a.k.a innerclass or closures or lambda
Unary>Binary>Keywords

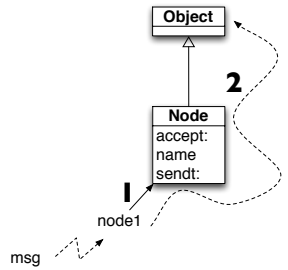
Goals

Syntax in a Nutshell
OO Model in a Nutshell

Instance and Class

- Only one model
- Uniformly applied
- Classes are objects too

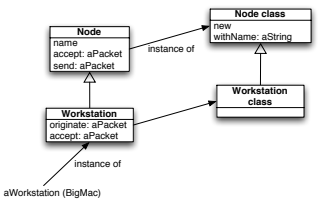
Lookup…Class + Inheritance



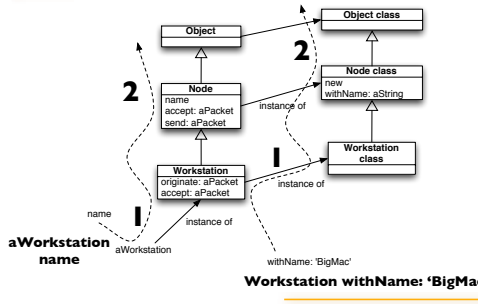
Classes are objects too

- Instance creation is just a message send to a ... Class
- Same method lookup than with any other objects
- a Class is the single instance of amanonymous class
  - Point is the single instance of Point class

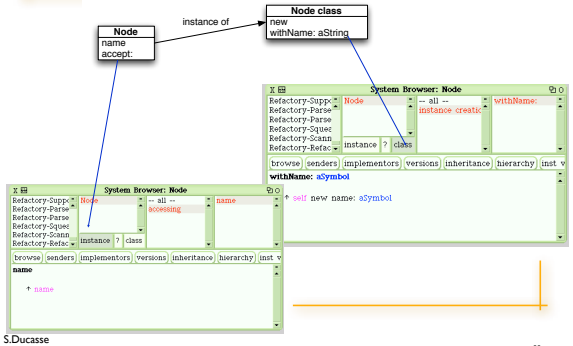
Class Parallel Inheritance



Lookup and Class Methods



About the Buttons



Summary

- Everything is an object
- One single model
- Single inheritance
- Public methods
- Protected attributes
- Classes are simply objects too
- Class is instance of another class
- One unique method lookup
  look in the class of the receiver

INRIA

RMod

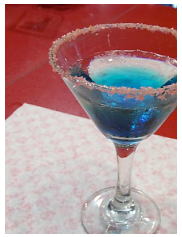
A little journey in a dynamic world

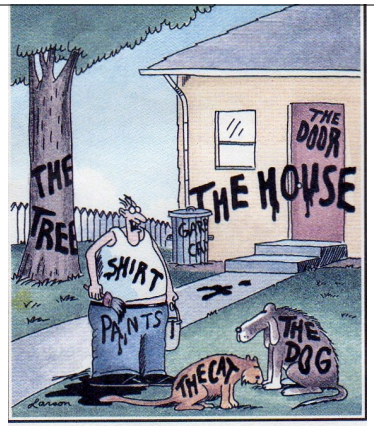
Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

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 + (...)

no Math.sin(0.7)
just 0.7 sin

Roadmap

Fun with numbers

Automatic coercion?

1 class

1 class

>SmallInteger

1 class maxVal

1 class maxVal

>1073741823

1 class maxVal + 1

1 class maxVal + 1

>1073741824

(1 class maxVal + 1) class

(1 class maxVal + 1) class

>LargePositiveInteger

(1/3) + (2/3)

(1/3) + (2/3)

>1

2/3 + 1

2/3 + 1

> 5/3

1000 factorial

1000 factorial

4023872600770937735437024339230039857193748642107146325437999104299385123986290205920442084869694048004799
8861019719605863166687299480855890132382966994459099742450408707375991882362772718873251977950595099527612
0874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432
5131853239584630755574091142624174743493475534286465766116677973966688202912073791438537195882498081268678
3837455973174613608537953452422158659320192809087829730843139284440328123155861103697680135730421616874760
9675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024
1536913982812648102130927612448963599287051149649754199093422215668325720808213331861168115536158365469840
4670897560290095053761647584772842188967964624494516076535340819890138544248798495995331910172335555660213
9450399736280750137837615307127761926849034352625200015888535147331611702103968175921510907788019393178114
1945452572238655414610628921879602238389714760885062768629671466746975629112340824392081601537808898939645
1826324367161676217916890977991190375403127462228998800519544441428201218736174599264295658174662830295557
0299024324153181617210465832036786906117260158783520751516284225540265170483304226143974286933061690897968
4825901254583271682264580665267699586526822728070757813918581788896522081643483448259932660433676601769996
1283186078838615027946595513115655203609398818061213855860030143569452722420634463179746059468257310379008
4024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346962715
4228458623773875382304838656889764619273838149001407673104466402598994902222217659043399018860185665264850
6179970235619389701786004081188972991831102117122984590164192106888438712185564612496079872290851929681937
2388642614839657382291123125024186649353143970137428531926649875337218940694281434118520158014123344828015
0513996942901534830776445690990731524332782882698646027898643211390835062170950025973898635542771967428222
4875758676575234422020757363056949882508796892816275384886339690995982628095612145099487170124451646126037
9029309120889086942028510640182154399457156805941872748998094254742173582401063677404595741785160829230135
3580818400969963725242305608559037006242712434169090041536901059339838357779394109700277534720000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000

1000 factorial / 999 factorial

1000 factorial / 999 factorial

> 1000

Roadmap

A first program

$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

12 printString

> ’12’

Strings are collections of chars

‘Tiramisu’ at: 1

Strings are collections of chars

‘Tiramisu’ at: 1

> $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
```

Syntax Summary

comment:	"a comment"
character:	$c $h $a $r $a $c $t $e $r $s $# $@
string:	'a nice string' 'lulu' 'l''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 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

```
factorial
	"Answer the factorial of the receiver."

	self = 0 ifTrue: [^ 1].
	self > 0 ifTrue: [^ self * (self - 1) factorial].
	self error: 'Not valid for negative integers'
```


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

1 class

1 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!

1 class
Date today

We sent messages to objects or classes!

1 class
Date today

Roadmap

Fun with binary messages

aReceiver aSelector anArgument

Used for arithmetic, comparison and logical operations
One or two characters taken from:
+ - / \ * ~ < > = @ % | & ! ? ,

1 + 2

1 + 2
>3

2 => 3

2 => 3
> false

10 @ 200

'Black chocolate' ,' is good'

Roadmap

Fun with keyword-based messages

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.keyword1keyword2(argument1, argument2)

receiver
keyword1: argument1
keyword2: argument2

equivalent to
receiver.keyword1keyword2(argument1, argument2)

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

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 :)

Conditions are messages sent to boolean

```
(x isBlue) ifTrue: [  ]
```

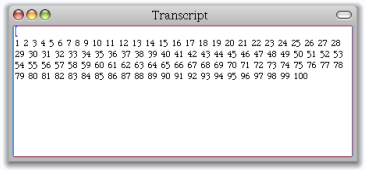
Roadmap

Fun with loops


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

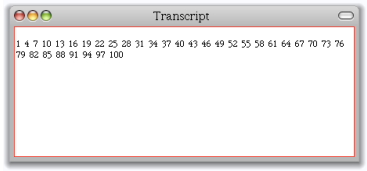


```
1 to: 100 do:
    [ :i | Transcript show: i ; space]
```



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

1 **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)

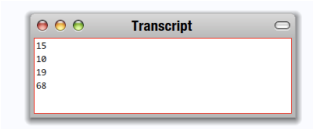
#(15 10 19 68) **collect:** [:i | i odd]

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
```

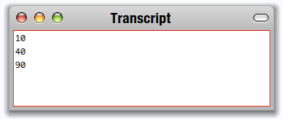

#(15 10 19 68) **do:**
[:i | Transcript show: i ; cr]

#(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."

  1 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:[1]

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:[1]

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:[1]

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]

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]
> 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?

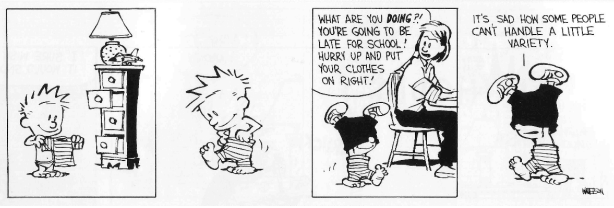
#() -> ''
#(a) ->'a'
#(a b c) -> 'a, b, c'

#(a b c)

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

#(a b c)

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



Smalltalk is fun

Pure simple powerful

Check the book
www.pharobyexample.org

www.seaside.st
(www.dabbledb.com)
www.pharo-project.org

INRIA

Objects to the Roots:
Learning from beauty

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

Really?!

No primitive types
No hardcoded constructs for conditional
Only messages
Only objects

and this works?
I mean really?
Not even slow?
Can't be real!

Motto

Let's open our eyes, look, understand, and deeply understand the underlying design aspects of object-oriented programming...

Booleans

```
3 > 0
    ifTrue: ['positive']
    ifFalse: ['negative']
```


Booleans

```
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:
...

Lazy Logical Operators

```
false and: [1 error: 'crazy']
```

PrIt-> false and not an error

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

But optimized by the compiler :)

Implementing not

Now you are good and you should implement it

Propose an implementation of not in a world where you do not have Booleans

false not -> **true**
true not -> **false**

Implementing ifTrue:ifFalse:

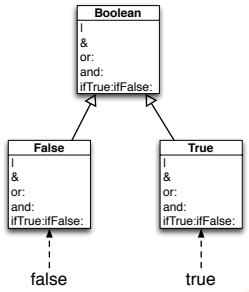
Now you are good and you should implement it

Propose an implementation of not in a world where you do not have Booleans

false ifTrue: [3] ifFalse: [5]
true ifTrue: [3] ifFalse: [5]

Boolean Objects

false and true are objects described by classes Boolean, True and False



Let's the receiver decide!

Boolean>>not

"Class Boolean is an abstract class that implements behavior common to true and false. Its subclasses are True and False. Subclasses must implement methods for logical operations &, not, controlling and:, or:, ifTrue:, ifFalse:, ifTrue:ifFalse:, ifFalse:ifTrue:"

Boolean>>not
"Negation. Answer true if the receiver is false, answer false if the receiver is true."
self subclassResponsibility

Not

false not -> **true**
true not -> **false**

Boolean>>not
"Negation. Answer true if the receiver is false, answer false if the receiver is true."
self subclassResponsibility

False>>not
"Negation -- answer true since the receiver is false."
^true
True>>not
"Negation--answer false since the receiver is true."
^false

| (Or)

- **true** | true -> **true**
- **true** | false -> **true**
- **true** | anything -> **true**

- false | **true** -> **true**
- false | **false** -> **false**
- false | **anything** -> **anything**

Boolean>> | aBoolean

Boolean>> | aBoolean
"Evaluating disjunction (OR). Evaluate the argument. Answer true if either the receiver or the argument is true."

self subclassResponsibility

False>> | aBoolean

false | **true** -> **true**
false | **false** -> **false**
false | **anything** -> **anything**

False>> | aBoolean
"Evaluating disjunction (OR) -- answer with the argument, aBoolean."

^ aBoolean

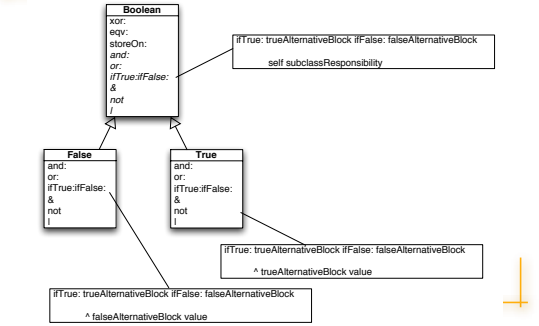
True>> | aBoolean

true | true -> **true**
true | false -> **true**
true | anything -> **true**

True>> | aBoolean
"Evaluating disjunction (OR) -- answer true since the receiver is true."

^ self

Boolean, True and False



Implementation Note

Note that the Virtual Machine shortcuts calls to boolean such as condition for speed reason.

Virtual machines such as VisualWorks introduced a kind of macro expansion, an optimisation for essential methods and Just In Time (JIT) compilation. A method is executed once and afterwards it is compiled into native code. So the second time it is invoked, the native code will be executed.

Ok so what?

You will probably not implement another Boolean classes

So is it really that totally useless?

Ternary logic

Boolean: true, false, unknown

A	*B*	*A* OR *B*	*A* AND *B*	NOT *A*
True	True	True	True	False
True	Unknown	True	Unknown	False
True	False	True	False	False
Unknown	True	True	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
False	True	True	False	True
False	Unknown	Unknown	False	True
False	False	False	False	True

More important...

Message sends act as case statements

OOP: the art of dispatching

Subclasses create your vocabulary

Avoid Conditional

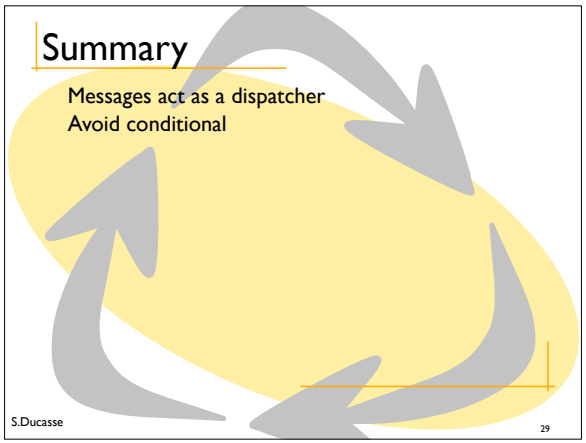
Use objects and messages

VM dispatch is a conditional switch: Use it!

AntiIfCampaign

Summary

Messages act as a dispatcher
Avoid conditional



INRIA

Inheritance Semantics and Method Lookup

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

Goal

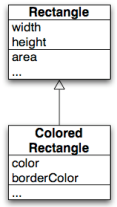
Inheritance
Method lookup
Self/super difference

Inheritance

Do not want to rewrite everything!
Often we want small changes
We would like to reuse and extend existing behavior

Solution: class inheritance

Each class defines or refines the definition of its ancestors



Inheritance

New classes
Can add state and behavior:
color, borderColor, borderWidth, totalArea

Can specialize ancestor behavior
intersect:

Can use ancestor's behavior and state
Can redefine ancestor's behavior
area to return totalArea

Inheritance in Smalltalk

Single inheritance

Static for the instance variables
At class creation time the instance variables are collected from the superclasses and the class. No repetition of instance variables.

Dynamic for the methods
Late binding (all virtual) methods are looked up at run-time depending on the dynamic type of the receiver.

Message Sending

receiver selector args

Sending a message = looking up the method that should be executed and executing it

Looking up a method: When a message (receiver selector args) is sent, the method corresponding to the message selector is looked up through the inheritance chain.

Method Lookup

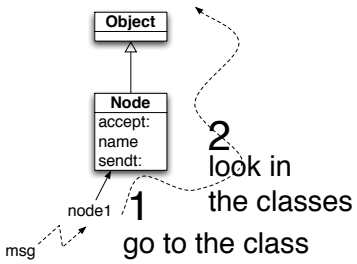
Two steps process

1: The lookup starts in the ***CLASS*** of the ***RECEIVER***.

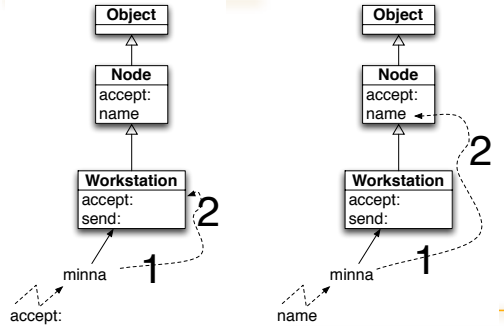
2: If the method is defined in the method dictionary, it is returned.

Otherwise the search continues in the superclasses of the receiver's class. If no method is found and there is no superclass to explore (class Object), this is an ERROR

Lookup: class and inheritance



Some Cases



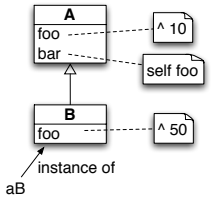
Method Lookup starts in Receiver Class

A new foo

B new foo

A new bar

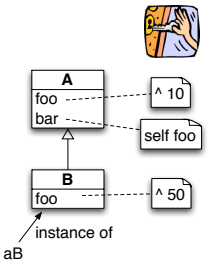
B new bar



Method Lookup starts in Receiver Class

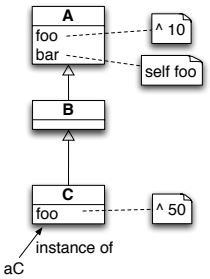
aB foo
(1) aB class => B
(2) Is foo defined in B?
(3) Foo is executed -> 50

aB bar
(1) aB class => B
(2) Is bar defined in B?
(3) Is bar defined in A?
(4) bar executed
(5) Self class => B
(6) Is foo defined in B
(7) Foo is executed -> 50



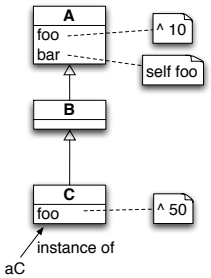
self **always** represents the receiver

- A new foo
- ->
- B new foo
- ->
- C new foo
- ->
- A new bar
- ->
- B new bar
- ->
- C new bar



self **always** represents the receiver

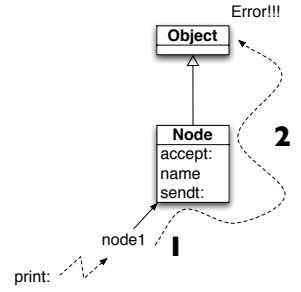
- A new foo
- -> 10
- B new foo
- -> 10
- C new foo
- -> 50
- A new bar
- -> 10
- B new bar
- -> 10
- C new bar



When message is not found

- If no method is found and there is no superclass to explore (class Object), a new method called #doesNotUnderstand: is sent to the receiver, with a representation of the initial message.

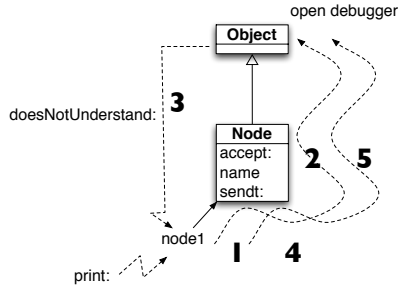
Graphically…



…in Smalltalk

- node1 print: aPacket
  - node is an instance of Node
  - print: is looked up in the class Node
  - print: is not defined in Node > lookup continues in Object
  - print: is not defined in Object => lookup stops + exception
  - message: node1 doesNotUnderstand: #(#print aPacket) is executed
  - node1 is an instance of Node so doesNotUnderstand: is looked up in the class Node
  - doesNotUnderstand: is not defined in Node => lookup continues in Object
  - doesNotUnderstand: is defined in Object => lookup stops + method executed (open a dialog box)

Graphically…



Roadmap

Inheritance
Method lookup
Self/super difference

How to Invoke Overridden Methods?

- Solution: Send messages to super
- When a packet is not addressed to a workstation, we just want to pass the packet to the next node, i.e., we want to perform the default behavior defined by Node.

```
Workstation>>accept: aPacket
    (aPacket isAddressedTo: self)
        ifTrue:[Transcript show: 'Packet accepted by the Workstation ',
self name asString]
        ifFalse: [super accept: aPacket]
```

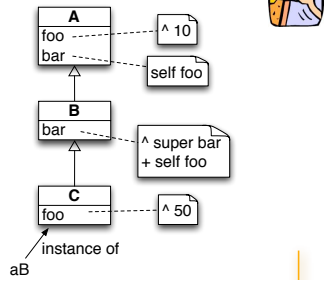
- Design Hint: Do not send messages to super with different selectors than the original one. It introduces implicit dependency between methods with different names.

The semantics of super

- Like self, ***super*** is a pseudo-variable that refers to the ***receiver*** of the message.
- It is used to invoke overridden methods.
- When using self, the lookup of the method begins in the class of the receiver.
- When using super, the lookup of the method begins in the ***superclass of the class of the method containing*** the super expression

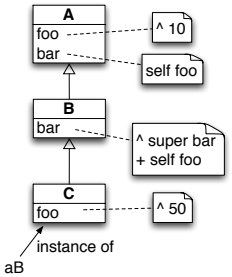
super *changes* lookup starting class

- A new foo
- A new bar
- B new foo
- B new bar
- C new foo
- C new bar



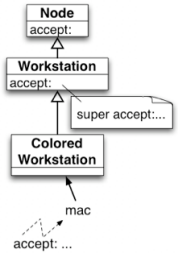
super *changes* lookup starting class

- A new bar
- -> 10
- B new bar
- -> 10 + 10
- C new bar
- -> 50 + 50



super is NOT the superclass of the receiver

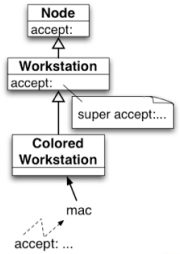
Suppose the WRONG hypothesis: *"The semantics of super is to start the lookup of a method in the superclass of the receiver class"*



super is NOT the superclass of the receiver

mac is instance of ColoredWorkStation
Lookup starts in ColoredWorkStation
Not found so goes up...

accept: is defined in Workstation
  lookup stops
  method accept: is executed
Workstation>>accept: does a super send
Our hypothesis: start in the super of the class of the receiver
=> superclass of class of a ColoredWorkstation is ... **Workstation !**



What you should know

- Inheritance of instance variables is made at class definition time.
- Inheritance of behavior is dynamic.
- self ****always**** represents the receiver.
- Method lookup starts in the class of the receiver.
- ***super*** represents the ***receiver*** but method lookup starts in the superclass of the class ***using*** it.
- ***Self is dynamic vs. super is static.***

Metaclasses in 7 Steps

Classes are objects too...
Classes are instances of other classes
...
One model applied twice

Metaclasses in 7 points

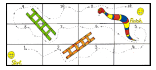
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

Adapted from Goldberg & Robson, *Smalltalk-80 — The Language*

Metaclasses in 7 points

1. **Every object is an instance of a class**
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

1. Every object is an instance of a class

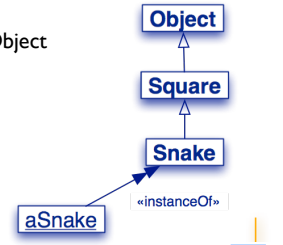


Metaclasses in 7 points

1. Every object is an instance of a class
2. **Every class eventually inherits from Object**
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass

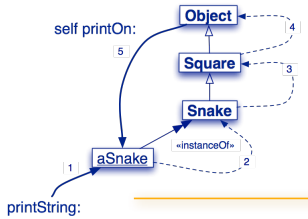
2. Every class inherits from Object

Every object is-an Object
The class of every object ultimately inherits from Object



The Meaning of is-a

When an object receives a message, the method is looked up in the method dictionary of its class, and, if necessary, its superclasses, up to Object



Responsibilities of Object

`Object`
represents the common object behavior
error-handling, halting …
all classes should inherit ultimately from `Object`

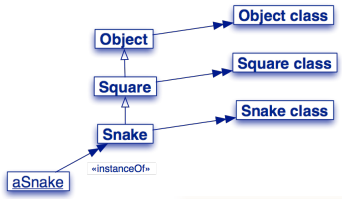
Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. **Every class is an instance of a metaclass**
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of

3. Every class is an instance of a metaclass

Classes are objects too!

Every class X is the unique instance of its metaclass, called X class



Metaclasses are implicit

There are no explicit metaclasses

Metaclasses are created implicitly when classes are created

No sharing of metaclasses (unique metaclass per class)

Metaclasses by Example

Square allSubclasses
Snake allSubclasses

Snake allInstances
Snake instVarNames

Snake back: 5

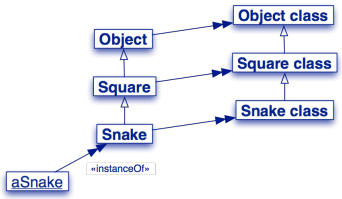
Snake selectors

Snake canUnderstand: #new

Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. **The metaclass hierarchy parallels the class hierarchy**
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of

4. The metaclass hierarchy parallels the



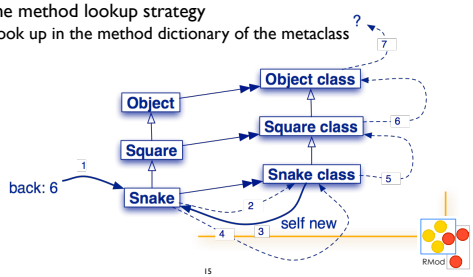
Uniformity between Classes and Objects

Classes are objects too, so …

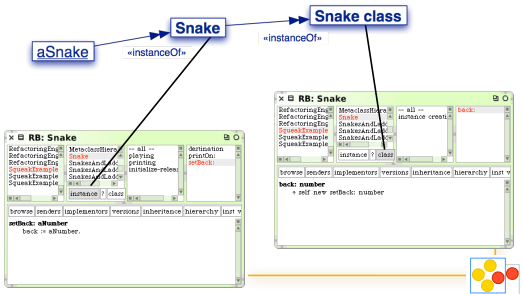
Everything that holds for objects holds for classes as well

Same method lookup strategy

Look up in the method dictionary of the metaclass



About the Buttons

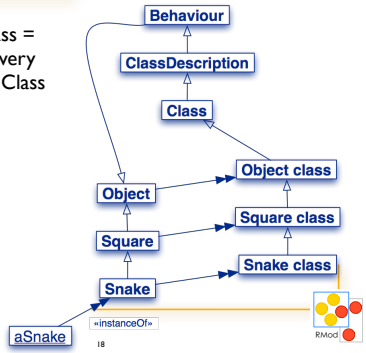


Metaclasses in 7 points

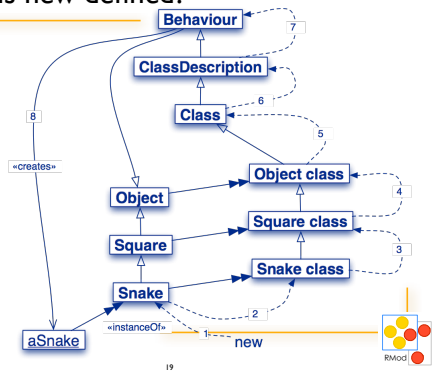
1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. **Every metaclass inherits from Class and Behavior**
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of

5. Every metaclass inherits from Class and

Every class is-a Class = The metaclass of every class inherits from Class



Where is new defined?



Responsibilities of Behavior

`Behavior`

Minimum state necessary for objects that have instances.
Basic interface to the compiler.

State:

class hierarchy link, method dictionary, description of instances (representation and number)

Methods:

creating a method dictionary, compiling method
instance creation (new, basicNew, new:, basicNew:)
class hierarchy manipulation (superclass:, addSubclass:)
accessing (selectors, allSelectors, compiledMethodAt:)
accessing instances and variables (allInstances, instVarNames)

Responsibilities of ClassDescription

`ClassDescription`

adds a number of facilities to basic Behavior:

named instance variables
category organization for methods
the notion of a name (abstract)
maintenance of Change sets and logging changes
most of the mechanisms needed for fileOut

`ClassDescription` is an abstract class: its facilities are intended for inheritance by the two subclasses, `Class` and `Metaclass`.

Responsibilities of Class

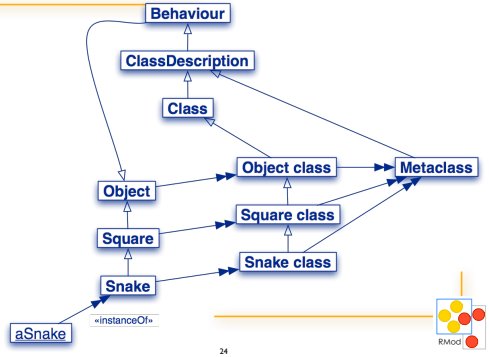
`Class`

represents the common behavior of all classes
name, compilation, method storing, instance variables …
representation for classVariable names and shared pool variables (`addClassVarName:`, `addSharedPool:`, `initialize`)
`Class` inherits from `Object` because `Class` is an `Object`
`Class` knows how to create instances, so all metaclasses should inherit ultimately from *`Class`*

Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. **Every metaclass is an instance of Metaclass**

6. Every metaclass is an instance of



Metaclass Responsibilities

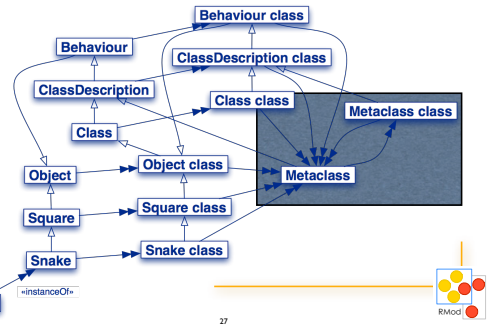
`Metaclass`

Represents common metaclass Behavior
instance creation (subclassOf:)
creating initialized instances of the metaclass's sole instance
initialization of class variables
metaclass instance protocol
(name:inEnvironment:subclassOf:....)
method compilation (different semantics can be introduced)
class information (inheritance link, instance variable, ...)

Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. **The metaclass of Metaclass is an**

7. The metaclass of Metaclass is an instance



Navigating the metaclass hierarchy

```
MetaclassHierarchyTest>>testHierarchy
  "The class hierarchy"
  self assert: Snake superclass = Square.
  self assert: Square superclass = Object.
  self assert: Object superclass superclass = nil. "skip ProtoObject"
  "The parallel metaclass hierarchy"
  self assert: Snake class name = 'Snake class'.
  self assert: Snake class superclass = Square class.
  self assert: Square class superclass = Object class.
  self assert: Object class superclass superclass = Class.
  self assert: Class superclass = ClassDescription.
  self assert: ClassDescription superclass = Behavior.
  self assert: Behavior superclass = Object.
  "The Metaclass hierarchy"
  self assert: Snake class class = Metaclass.
  self assert: Square class class = Metaclass.
  self assert: Object class class = Metaclass.
  self assert: Class class class = Metaclass.
  self assert: ClassDescription class class = Metaclass.
  self assert: Behavior class class = Metaclass.
  self assert: Metaclass superclass = ClassDescription.
  "The fixpoint"
  self assert: Metaclass class class = Metaclass.
```

INRIA

Elements of Design - Inheritance/Composition

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

A Formating Text Editor

With several possible algorithms

formatWithTex
formatFastColoring
formatSlowButPreciseColoring

Identify your own criterias

Define criterias to compare your solutions?

Code Smells


```
Composition>>repair
    formatting == #Simple
        ifTrue: [ self formatWithSimpleAlgo]
        ifFalse: [ formatting == #Tex
                ifTrue: [self formatWithTex]

                ....]
```


Inheritance?

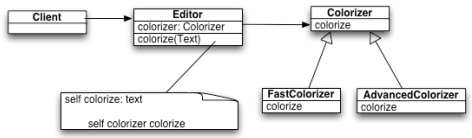
May not be the solution since:
- you have to create objects of the right class
- it is difficult to change the policy at run-time
- you can get an explosion of classes bloated with the use of a functionality and the functionalities.
- no clear identification of responsibility

Inheritance vs. Composition

Inheritance is not a panacea
- Require class definition
- Require method definition
- Extension should be prepared in advance
- No run-time changes

Ex: editor with spell-checkerS, colorizerS, mail-readerS….
- No clear responsibility
- Code bloated
- Cannot load a new colorizers

Delegating to other Objects



myEditor setColorizer: FastColorizer new.
myEditor setColorizer: AdvancedColorizer new.
Strategy design pattern

Composition Analysis

Pros
- Possibility to change at run-time
- Clear responsibility
- No blob
- Clear interaction protocol

Cons
- New class
- Delegation
- New classes