

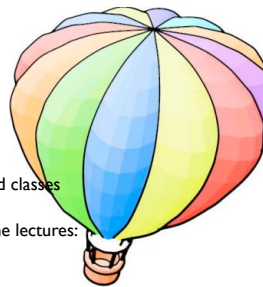


The Taste of Smalltalk

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
<http://www.iam.unibe.ch/~ducasse/>

Goals

- Two examples:
 - "hello world"
 - a LAN simulator
- To give you an idea of:
 - the syntax
 - the elementary objects and classes
 - the environment
- To provide the basis for all the lectures:
 - all the code examples,
 - constructs,
 - design decisions, ...



An Advice

You do not have to know everything!!!

- "Try not to care - Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how a thing works before they can use it. This means it takes quite a while before they can master Transcript show: 'Hello World'. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".... Alan Knight. Smalltalk Guru
- We will show you how to learn and find your way



Some Conventions

- Return Values
 - 1 + 3 -> 4
 - Node new -> aNode
- Method selector #add:
- Method scope conventions
- Instance Method defined in class Node:
 - Node>>accept: aPacket
- Class Method defined in class Node (in the class of the class Node)
 - Node class>>withName: aSymbol
- aSomething is an instance of the class Something



Roadmap

- "hello world"
- Syntax
- a LAN simulator



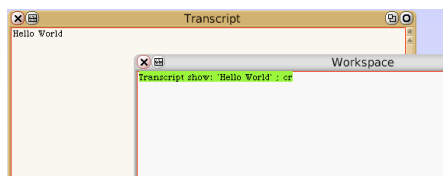
Hello World

Transcript show: 'hello world'

- At anytime we can dynamically ask the system to evaluate an expression. To evaluate an expression, select it and with the middle mouse button apply doIt.
- **Transcript** is a special object that is a kind of standard output.
- It refers to a TextCollector instance associated with the launcher.



Transcript show: 'hello world'



Everything is an Object

The workspace is an object.
 The window is an object: it is an instance of ApplicationWindow.
 The text editor is an object: it is an instance of ParagraphEditor.
 The scrollbars are objects too.
 'hello world' is an object: it is aString instance of String.
 #show: is a Symbol that is also an object.
 The mouse is an object.
 The parser is an object: instance of Parser.
 The compiler is also an object: instance of Compiler.
 The process scheduler is also an object.
 The garbage collector is an object: instance of MemoryObject.
 Smalltalk is a consistent, uniform world written in itself. You can learn how it is implemented, you can extend it or even modify it. All the code is available and readable



Smalltalk Object 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



Roadmap

- Hello World
- First look at the syntax
- LAN Simulator



S.Ducasse

10



Complete Syntax on a PostCard

exampleWithNumber: x
"Illustrates every part of Smalltalk method syntax. It has unary, binary, and key word messages, declares arguments and 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."

```
|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
```

S.Ducasse

11



Yes ifTrue: is sent to a boolean

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

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

S.Ducasse

12



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 perform the loop on itself

S.Ducasse

13



Dolt, PrintIt, InspectIt and Accept

- **Accept = Compile:** Accept a method or a class definition
- **Dolt:** send a message to an object
- **PrintIt:** send a message to an object + print the result (#printOn:)
- **InspectIt:** send a message to an object + inspect the result (#inspect)

S.Ducasse

14



Objects send messages

- Transcript show: 'hello world'
- The above expression is a message
- the object Transcript is the **receiver** of the message
- the **selector** of the message is #show:
- one **argument:** a string 'hello world'
- Transcript is a global variable (starts with an uppercase letter) that refers to the Launcher's report part.

S.Ducasse

15



Vocabulary Point

Message passing or sending a message is equivalent to invoking a method in Java or C++
calling a procedure in procedural languages
applying a function in functional languages
of course the last two points must be considered under the light of polymorphism

S.Ducasse

16



Roadmap

- Hello World
- First look at the syntax
- **LAN Simulator**



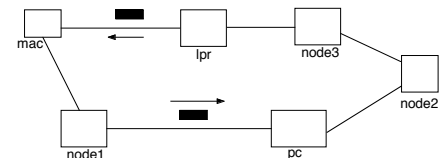
S.Ducasse

17



A LAN Simulator

A LAN contains nodes, workstations, printers, file servers. Packets are sent in a LAN and each node treats them differently.



S.Ducasse

18

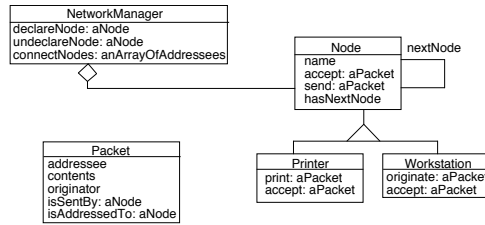


Three Kinds of Objects

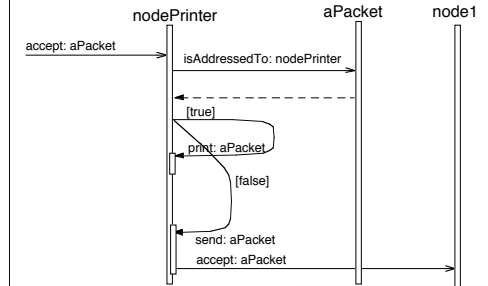
Node and its subclasses represent the entities that are connected to form a LAN.
 Packet represents the information that flows between Nodes.
 NetworkManager manages how the nodes are connected



LAN Design



Interactions Between Nodes



Node and Packet Creation

```

[macNode pcNode node1 printerNode node2 node3 packet]
macNode := Workstation withName: #mac.
pcNode := Workstation withName: #pc.
node1 := Node withName: #node1.
node2 := Node withName: #node2.
node3 := Node withName: #node2.
printerNode := Printer withName: #lpr.
macNode nextNode: node1.
node1 nextNode: pcNode.
pcNode nextNode: node2.
node3 nextNode: printerNode.
lpr nextNode: macNode.

packet := Packet send: 'This packet travelled to' to: #lpr.
    
```



Objects Send Messages

Message: **1 + 2**
 receiver: 1 (an instance of SmallInteger)
 selector: #+
 arguments: 2

Message: **lpr nextNode: macNode**
 receiver: lpr (an instance of LanPrinter)
 selector: #nextNode:
 arguments: macNode (an instance of Workstation)

Message: **Packet send: 'This packet travelled to' to: #lpr**
 receiver: Packet (a class)
 selector: #send:to:
 arguments: 'This packet travelled to' and #lpr



Transmitting a Packet

```

[aLan packet macNode]
...
macNode := aLan findNodeWithAddress: #mac.
packet := Packet send: 'This packet travelled to the printer' to: #lpr.
macNode originate: packet.
-> mac sends a packet to pc
-> pc sends a packet to node1
-> node1 sends a packet to node2
-> node2 sends a packet to node3
-> node3 sends a packet to lpr
-> lpr is printing
-> this packet travelled to lpr
    
```



How to Define a Class?

- Fill the template:
 NameOfSuperclass subclass: #NameOfClass
 instanceVariableNames: 'instVarName1'
 classVariableNames: 'ClassVarName1 ClassVarName2'
 poolDictionaries: ''
 category: 'LAN'



Packet

- For example to create the class Packet
Object subclass: #Packet
 instanceVariableNames: 'addressee originator'
 contents '
 classVariableNames: ''
 poolDictionaries: ''
 category: 'LAN'



How to Define a Method?

message selector and argument names
 "comment stating purpose of message"
 | temporary variable names |
 statements

accept: thePacket

"If the packet is addressed to me, print it. Otherwise just behave like a normal node."

```

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



In Java

- In Java we would write
void accept(thePacket Packet)
/*If the packet is addressed to me, print it. Otherwise just
behave like a normal node.*/

```
if (thePacket.isAddressedTo(this)){      this.print  
(thePacket)}  
    else super.accept(thePacket)}
```



Summary

- What is a message?
- What is the message receiver?
- What is the method selector?
- How to create a class?
- How to define a method?