Object-Oriented Design with Smalltalk

A Pure Object Language and its Environment

Dr. Stéphane Ducasse

2002

Table of Contents

March 18, 2002

1. Introduction

- Lecture:
  Object-Oriented Design with Smalltalk -
  A Pure Object Language and its environment
- by:

  - Dr. Stéphane Ducasse
    Schuetzenmattstrasse 14 / Room 101 / Tel. +41 31 631 4903
    ducasse@iam.unibe.ch - http://www.iam.unibe.ch/~ducasse/
  - Michele Lanza
    Schuetzenmattstrasse 14 / Room 106 / Tel. +41 31 631 4868
    lanza@iam.unibe.ch - http://www.iam.unibe.ch/~lanza/
  - Prof. Dr. Oscar Nierstrasz
    Schuetzenmattstrasse 14 / Room 103 / Tel. +41 31 631 4618
    oscar@iam.unibe.ch - http://www.iam.unibe.ch/~oscar/
  - Dr. Roel Wuyts
    Schuetzenmattstrasse 14 / Room 102 / Tel. +41 31 631 3314
    wuyts@iam.unibe.ch - http://www.iam.unibe.ch/~wuyts/

Structure of this Lecture

- Introduction
- Part I - Basic Smalltalk Elements
  - History and Concepts
  - Quick Overview of the Environment
  - A Taste of Smalltalk
  - Smalltalk Syntax in a Nutshell
  - Syntax and Messages
  - Dealing with Classes
  - Basic Objects, Conditionals and Loops
  - Numbers
  - Exceptions
  - Streams

Structure of this Lecture (II)

- Part II - Advanced Smalltalk Elements
  - Advanced Classes
  - The Model-View-Controller Paradigm
  - Processes and Concurrency
  - Classes and Metaclasses - an Analysis
  - Common Mistakes and Debugging
  - The Internal Structure of Objects
  - Blocks and Optimization
  - Advanced Blocks
- Part III - Design Considerations
  - Abstract Classes
  - Elements of Design
  - Elementary Design Issues
  - Selected Idioms
  - Selected Design Patterns

Structure of this Lecture (III)

- Part IV - Comparisons
  - Comparing C++, Java and Smalltalk
  - Smalltalk for the Java Programmer
  - Smalltalk for the Ada Programmer
- References

Web Resources

Local Website	http://www.iam.unibe.ch/~scg/Resources/Smalltalk/
Steph’s Website	http://www.iam.unibe.ch/~ducasse/PubHTML/Smalltalk.html
Cincom Smalltalk	http://www.cincom.com
Squeak	http://www.squeak.org
Dolphin Smalltalk	http://www.object-arts.com/Home.htm
STIC	http://www.stic.org
	http://www.smalltalk.org
	http://www.goodstart.com/index.shtml
	http://st-www.cs.uiuc.edu/
VisualWorks Wiki:	http://brain.cs.uiuc.edu/VisualWorks/
VisualAge Wiki:	http://brain.cs.uiuc.edu/VisualAge/
Local Wiki:	http://scgwiki.iam.unibe.ch:8080/SmalltalkWiki/
Newsgroup:	comp.lang.smalltalk
ESUG	http://www.esug.org
BSUG	http://www.bsug.org
GSUG	http://www.gsug.org
SSUG	http://www.iam.unibe.ch/~ssug/

About this lecture...

- If you have problems or questions, *ask!*
  - Ignorance is not always bliss...
- Grab one of the Smalltalk distributions and play with it.
  - We suggest:

    VisualWorks 5i from www.cincom.com or http://brain.cs.uiuc.edu/
    Squeak from www.squeak.org
- Do the exercises!!!
- Authors:
  - Stéphane Ducasse
  - Juan Carlos Cruz, Michele Lanza, Oscar Nierstrasz, Matthias Rieger

Part I - Basic Smalltalk Elements

- History and Concepts
- Quick Overview of the Environment
- A Taste of Smalltalk
- Smalltalk Syntax in a Nutshell
- Syntax and Messages
- Dealing with Classes
- Basic Objects, Conditionals and Loops
- Numbers
- Exceptions
- Streams

2. History and Concepts

- History
- Context
- Run-Time Architecture
- Concepts

Smalltalk - A State of Mind

- A small and uniform language
  - Syntax fits on one sheet of paper
- A large set of reusable classes
  - Basic Data Structures, GUI classes, Database Access, Internet, Graphics
- A set of powerful development tools
  - Browsers, GUI Builders, Inspectors, Change Management Tools, Crash Recovery Tools, Project Management Tools
- A run-time environment based on virtual machine technology
  - Platform Independent
- Envy
  - Team Working Environment (releasing, versioning, deploying).

Smalltalk - The Inspiration

"Making simple things very simple and complex things very possible."

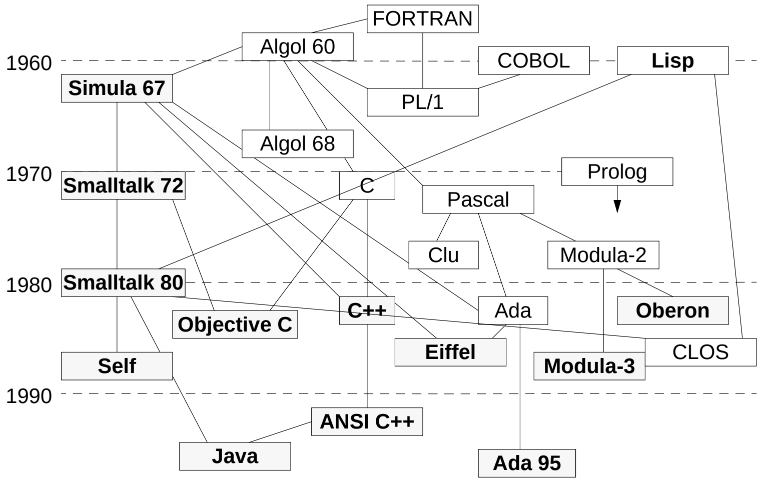
Alan Kay

- Flex (Alan Kay, 1969)
- Lisp (Interpreter, Blocks, Garbage Collection)
- Turtle graphics (The Logo Project, Programming for Children)
- Direct Manipulation Interfaces (Sketchpad, Alan Sutherland, 1960)
- NLS, (Doug Engelbart, 1968), “the augmentation of human intellect”
- Simula (Classes and Message Sending)
  - Description of real Phenomenons by means of a specification language
    -> modelling
- Xerox PARC (Palo Alto Research Center)
  - DynaBook: a Laptop Computer for Children

The Precursor, The Innovator & The Visionary

- First to be based on Graphics
  - Multi-Windowing Environment (Overlapping *Windows*)
  - Integrated Development Environment
    -> Debugger, Compiler, Text Editor, Browser
  - With a pointing Device
    -> Yes, a Mouse
- Ideas were taken over
  - Apple Lisa, Mac
  - Microsoft Windows 1.0
- Virtual Machine -> Platform independent
- Garbage Collector -> Time for some real thinking...
- Just in Time Compilation
- Everything was there, the complete Source Code

The History



The History (II)

- Internal
  - 1972 - First Interpreter -> More Agents than Objects (every object can specify its own syntax)
  - 1976 - Redesign -> A Hierarchy of classes with a Unique Root, Fixed Syntax, Compact Byte Code, Contexts, Processes, Semaphores, Browsers, GUI Library. Projects: ThingLab, Visual Programming Environment Programming by Rehearsal.
  - 1978 - NoteTaker Project, Experimentation with 8086 Microprocessor with only 256 KB RAM.
- External
  - 1980 - Smalltalk-80 (ASCII, cleaning primitives for portability, Metaclasses, Blocks as first-class Objects, MVC). Projects: Gallery Editor (mixing text, painting and animations) + Alternate Reality Kit (physics simulation)
  - 1981 - Books + 4 external virtual machines (Dec, Apple, HP and Tektronix) -> GC by generation scavenging
  - 1988 - Creation of Parc Place Systems
  - 1992 - ANSI Draft
  - 1995 - New Smalltalk implementations (MT, Dolphin, Squeak)
  - 2000 - Things are moving again...

Smalltalk's Concepts

- *Everything is an object* (numbers, files, editors, compilers, points, tools, boolean).
- Objects communicate *only* by message passing.
- Each object is an instance of one class (which is also an object).
- A class defines the structure and the behaviour of its instances.
- Each object possesses its own set of values.
- Dynamic Typing.
- Purely based on *late binding*.

- Programming in Smalltalk: *Reading and writing an interactive Book*
  - Reading the interface of the classes: (table of contents of a book)
  - Understanding the way the classes are implemented: (reading the chapters)
  - Extending and changing the contents of the system: (writing into the book)

Messages, Methods and Protocols

- Message: **What** behaviour to perform

```
aWorkstation accept: aPacket
```

- Method: **How** to carry out the behaviour

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

- Protocol: The complete set of messages an object responds to:

```
#name #initialize #hasNextNode #connectedTo: #name: #nextNode #nextNode: #printOn: #simple-PrintString #typeName #accept: #send:
```

- Often grouped into categories:

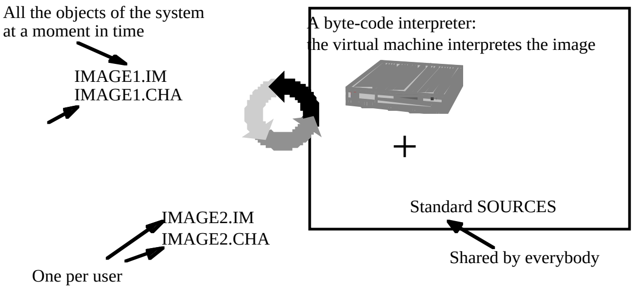
```
accessing           #name
initialize-release  #initialize
testing             #hasNextNode
connection          #connectedTo:
private             #name: #nextNode #nextNode:
printing            #printOn: #simplePrintString #typeName
send-receive        #accept: #send:
```


Objects, Classes and Metaclasses

- Every object is an instance of a class
- A class specifies the structure and the behaviour of all its instances
- Instances of a class share the same behaviour and have a specific state
- Classes are objects that create other instances
- Metaclasses are classes that create classes as instances
- Metaclasses describe class behaviour and state (subclasses, method dictionary, instance variables...)

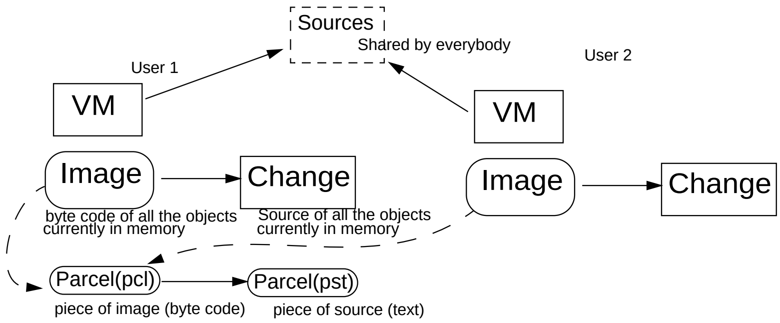
Smalltalk Run-Time Architecture

- ❑ Virtual Machine + Image + Changes and Sources



- ❑ The byte-code is in fact translated into native code by a just-in-time compiler.
- ❑ The source and the changes are not necessary for interpreting the byte-code, this is just for the development. Normally they are removed for deployment.
- ❑ An application can be delivered as some byte-code files that will be executed with a VM. The development image is stripped to remove the unnecessary development components.

VisualWorks Smalltalk Run-Time Architecture



- ❑ Parcels reproduce the schema of the image and change:
  *.pcl are the byte code, *.pst are the source code
- ❑ Parcels allows for fast atomic loading/unloading and prerequisite parcels
- ❑ Good for dynamic loading and source code management

3. Quick Overview of the Environment

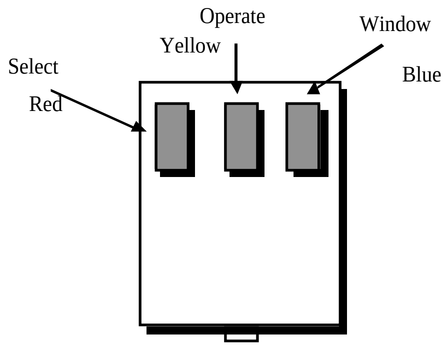
The following screenshots are taken from VisualWorks 2.5.

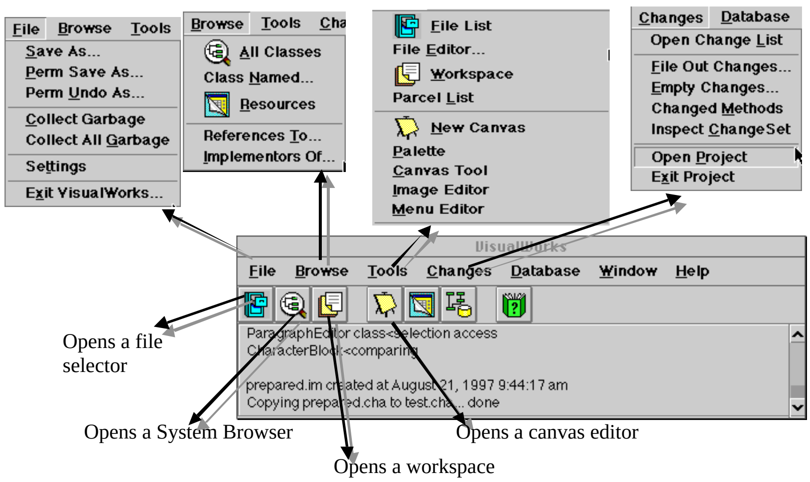
In the meantime VisualWorks has reached version 7.0.

Several other Smalltalk dialects have state-of-the-art GUIs, but a nice GUI is not the point:

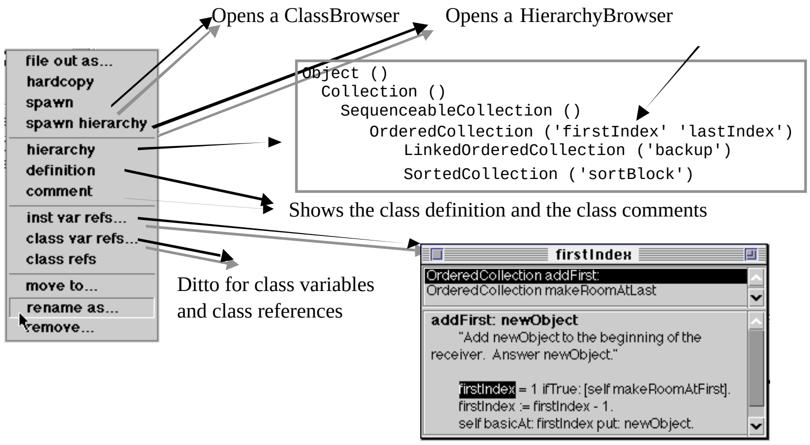
Try to look beyond the Facade...

Mouse Semantics

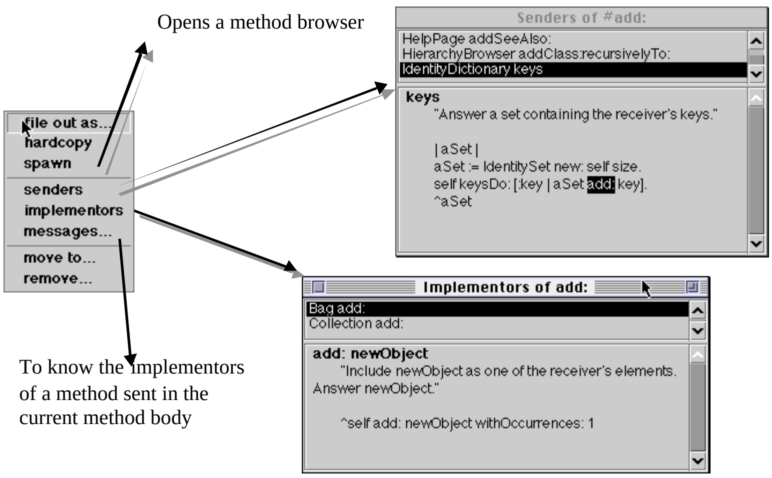




Class MenuBar

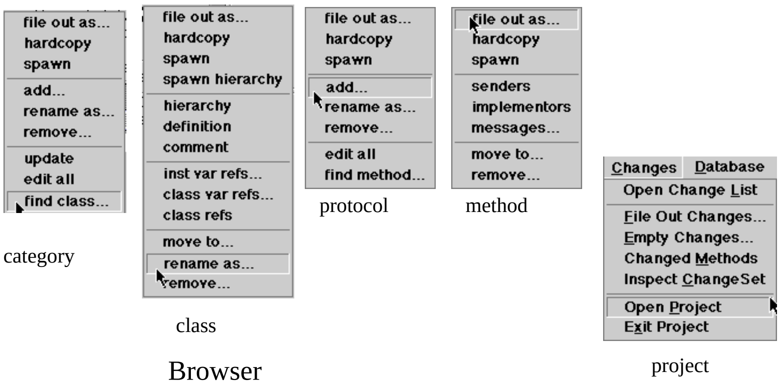


Method MenuBar

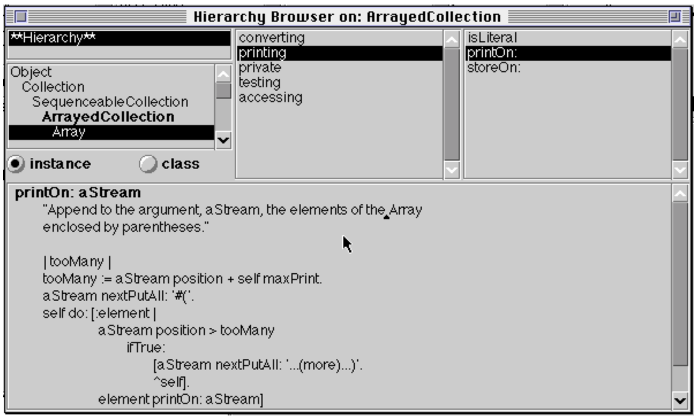


Cross Reference Facilities

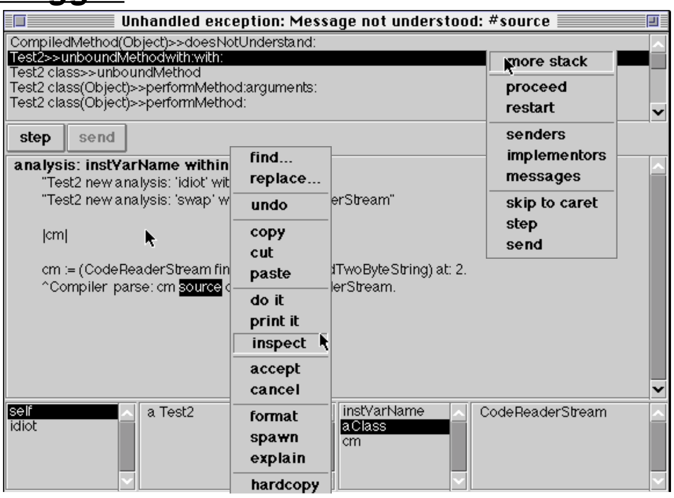
Filing Out



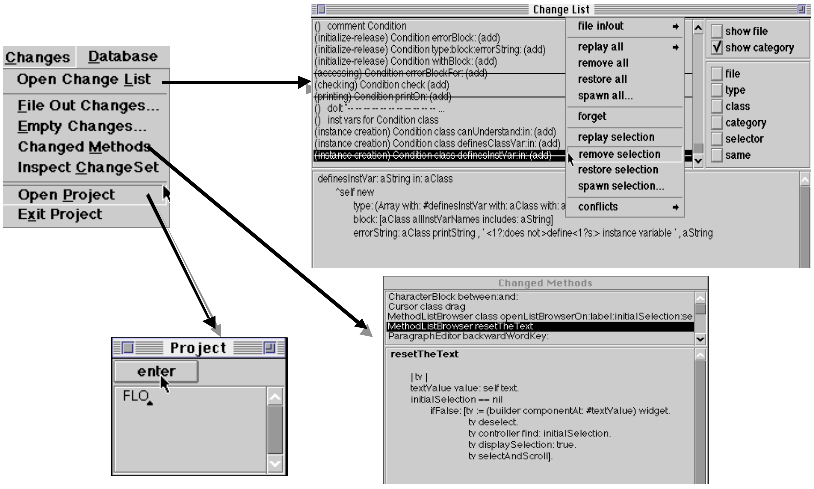
Hierarchy Browser



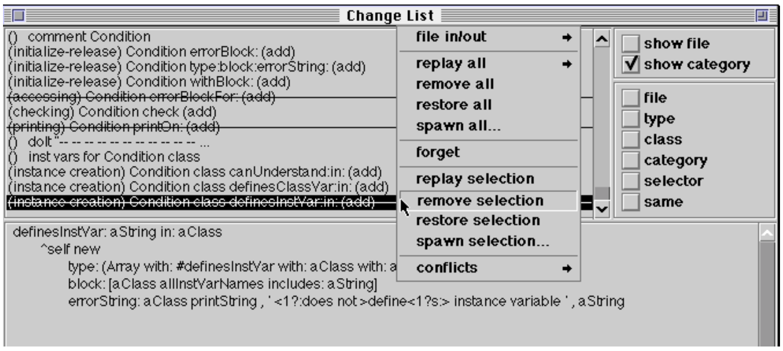
Debugger



Crash Recovery

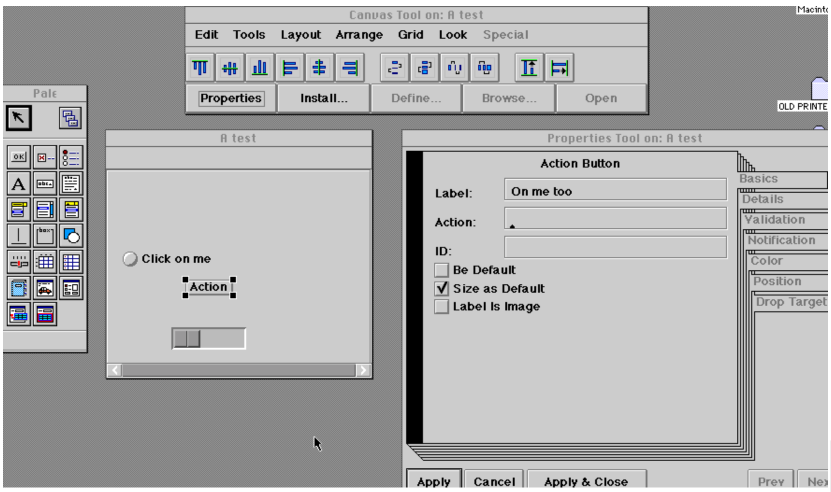


Condensing Changes



SourceFileManager new condenseChanges

UIBuilder



4. A Taste of Smalltalk

"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, registered Guru

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, ...

Power & Simplicity: The Syntax on a PostCard

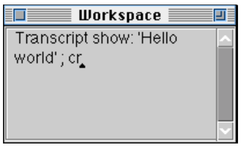
- From Ralph Johnson

```
exampleWithNumber: x
"This is a small method that illustrates every part of Smalltalk method syntax
except primitives, which aren't very standard. 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
```


Some Conventions

- Code `Transcript show: 'Hello world'`
- Return Values
  - `1 + 3 ->` *4*
  - `Node new ->` *aNode*
  - `Node new PrIt ->` *a Workstation with name:#pc connectedto:#mac*
- 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 the class `Node`)
    - **Node class>>**`withName: aSymbol`
- `aSomething` is an instance of the class `Something`
- DoIt, PrintIt, InspectIt and Accept
  - Accept = Compile: Accept a method or a class definition
  - DoIt = 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`)

Hello World




```
Transcript show: 'hello world'
```

- During implementation, we can dynamically ask the interpreter 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.

Everything is an Object

- The launcher is an object.
- The icons are objects.
- 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 word'` is an object: it is a`String` 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. Almost all the code is available and readable -> Book concept.

Objects communicate via Messages

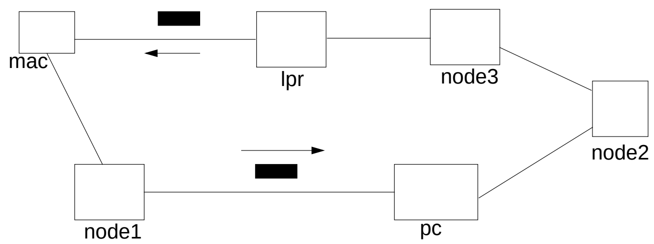
```
Transcript show: 'hello world'
```

- 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.

- Vocabulary Concerns: 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

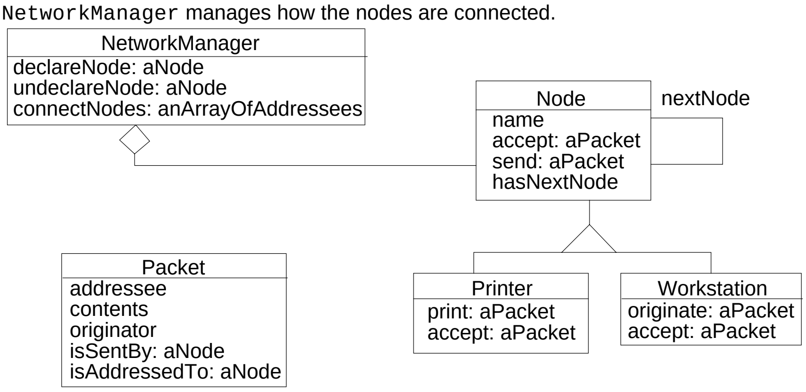
A LAN Simulator

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

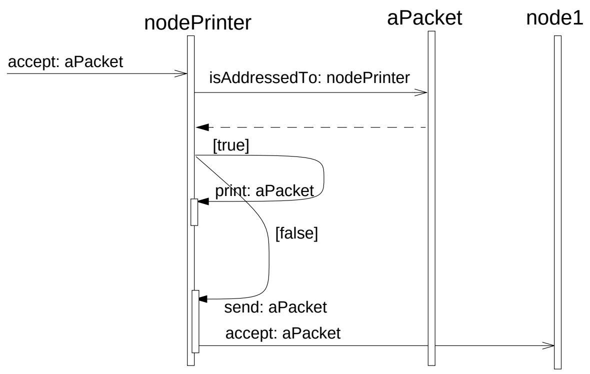


Three Kinds of Objects

`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.



Interactions Between Nodes



Node and Packet Creation

```
|macNode pcNode node1 printerNode node2 node3 packet|
"nodes definition"
macNode := Workstation withName: #mac.
pcNode := Workstation withName: #pc.
node1 := Node withName: #node1.
node2 := Node withName: #node2.
node3 := Node withName: #node2.
printerNode := Printer withName: #lpr.
"Node connections"
macNode nextNode: node1.
node1 nextNode: pcNode.
pcNode nextNode: node2.
node3 nextNode: printerNode.
lpr nextNode: macNode.

"packet creation"
packet := Packet send: 'This packet travelled to the printer' to: #lpr.
```


Objects communicate via Messages (II)

- 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 the printer' to: #lpr
  - receiver: Packet (a class)
  - selector: #send:to:
  - arguments: 'This packet travelled to the printer' and #lpr
- Message: Workstation withName: #mac
  - receiver: Workstation (a class)
  - selector: #withName:
  - arguments: #mac

The Definition of a LAN

- To simplify the creation and the manipulation of a LAN:

```
| aLan |
aLan := NetworkManager new.
aLan createAndDeclareNodesFromAddresses: #(node1 node2 node3) ofKind: Node.
aLan createAndDeclareNodesFromAddresses: #(mac pc) ofKind: Workstation.
aLan createAndDeclareNodesFromAddresses: #(lpr) ofKind: LanPrinter.

aLan connectNodesFromAddresses: #(mac node1 pc node2 node3 lpr)
```

- Now we can query the LAN to get some nodes:

```
aLan findNodeWithAddress: #mac
```

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 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'LAN'
```

- For example to create the class Packet

```
Object subclass: #Packet
   instanceVariableNames: 'addressee originator contents '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LAN'
```


How to Define a Method

- Follow the template:

```
message selector and argument names
"comment stating purpose of message"

| temporary variable names |
statements

LanPrinter>>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 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
```

5. Smalltalk Syntax in a Nutshell

- From Ralph Johnson:

```
exampleWithNumber: x
"This is a small method that illustrates every part of a Smalltalk method syntax
except primitives, which aren't very standard. It has unary, binary, and key
word messages, declares arguments and temporaries (but not block temporaries),
accesses a global variable (but not an instance variable), uses literals (array,
character, symbol, string, integer, float), uses the pseudo variables 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 VW5i)
;	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 in a Nutshell

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:	(...)

Messages instead of a predefined Syntax

- In Java, C, C++, Ada constructs like >>, if, for, etc. are hardcoded into the grammar
- In Smalltalk there are just messages defined on objects

(>>) bitShift: is just a message sent to numbers

```
10 bitShift: 2
```

(if) ifTrue: is just messages sent to a boolean

```
(1> x) ifTrue:
```

(for) do:, to:do: are just messages to collections or numbers

```
#(a b c d) do: [:each | Transcript show: each ; cr]
1 to: 10 do: [:i | Transcript show: each printString; cr]
```

-> Minimal parsing
-> Language is extensible

Class and Method Definition Revisited

- Class Definition: A message sent to another class

```
Object subclass: #Node
    instanceVariableNames: 'name nextNode'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'LAN'
```

  - Instance variables are instance-based protected
- Method: Normally done in a browser or (by directly invoking the compiler)

```
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]
```

  - Methods are public

Instance Creation

- 1, ‘abc’
- Basic class creation messages are

```
new, new:, basicNew, basicNew:

Packet new
```

- Class specific message creation

```
Workstation withName: #mac
```

6. Syntax and Messages

- The syntax of Smalltalk is simple and uniform, but it can look strange at first sight!
  - Literals: numbers, strings, arrays....
  - Variable names
  - Pseudo-variables
  - Assignments, returns
  - Message Expressions
  - Block expressions
- Read it as a non-computer-literate person:

```
| bunny |
bunny := Actor fromFile: 'bunny.vrml'.
bunny head doEachFrame:
[ bunny head
   pointAt: (camera transformScreenPointToScenePoint: (Sensor mousePoint) using: bunny)
   duration: camera rightNow ]
```


Literals, an Overview

- Numbers:
  - SmallInteger, Integer,
    `4, 2r100 (4 in base 2),3r11 (4 in base 3), 1232`
  - Fraction, Float, Double
    `3/4, 2.4e7, 0.75d`
- Characters:
  `$F, $Q $U $E $N $T $i $N`
- Unprintable characters:
  `Character space, Character tab, Character cr`
- Symbols:
  `#class #mac #at:put:  #+ #accept:`
- Strings:

```
#mac asString -> 'mac'
12 printString -> '12'
'This packet travelled around to the printer' 'l''idiot'
String with: $A
```

  - To introduce a single quote inside a string, just double it.

Literals, an Overview (II)

- Arrays:

```
#(1 2 3) #('lulu' (1 2 3))  #('lulu' #(1 2 3))
#(mac node1 pc node2 node3 lpr) an array of symbols.
When one prints it it shows #(#mac #node1 #pc #node2 #node3 #lpr)
```

  - Byte Array:
    `#[1 2 255]`
- Comments:
  `"This is a comment"`
  - A comment can span several lines. Moreover, avoid putting a space between the “ and the first letter. When there is no space, the system helps you to select a commented expression. You just go after the “ character and double click on it: the entire commented expression is selected. After that you can printIt or doIt, etc.

Literals, the Arrays

- Heterogenous

```
#('lulu' (1 2 3)) PrIt-> #('lulu' #(1 2 3))
#('lulu' 1.22 1) PrIt-> #('lulu' 1.22 1)
```

- An array of symbols:

```
#(calvin hobbes suzie) PrIt-> #(#calvin #hobbes #suzie)
```

- An array of strings:

```
#('calvin' 'hobbes' 'suzie') PrIt->  #('calvin' 'hobbes' 'suzie')
```

- Only the creation time differs between literal arrays and arrays. Literal arrays are known at compile time, arrays at run-time.

```
#(Packet new) an array with two symbols and not an instance of Packet
Array new at: 1 put: (Packet new) is an array with one element an instance of Packet
```

- Literal or not

```
#(...) considers elements as literals and false true and nil
#( 1 + 2 ) PrIt-> #(1 #+ 2)
Array with: (1 +2) PrIt->  #(3)
```


Literals, the Arrays (II)

- Implementation dependent technical note: Literal arrays may only contain literal objects, `false`, `true` and `nil`

```
'mac' asArray is an array of character
(#(false true nil) at: 2 )
   ifTrue:[ Transcript show: ‘this is really true’]
   ifFalse: [ 1/0]
```

- Literature (the Goldberg book) defines a *literal* as an object whose value always refers to the same object. This is a first approximation to present the concept. However, if we examine literals according to this principle, this is false in VisualWorks (VisualAge has a safer definition.)
- Other Literature defines literals as numbers, characters, strings of characters, arrays, symbols, and two strings, floats , arrays, but they do not refer (hopefully) to the same object.
- In fact literals are objects created at compile-time or even already exist in the system and are stored into the compiled method literal frame. A compiled method is an object that holds the bytecode translation of the source code. The literal frame is part of a compiled method that stores the literals used by the methods. To see it do: *Point inspect ->methodDict-> aCompiledMethod*

Literals, the Arrays (III)

- The following example illustrates the difference between a literal array and a newly created instance of Array created via Array new:. Let us define the following method:

```
SmallInteger>>m1
|anArray|
anArray := #(nil).
(anArray at: 1 ) isNil ifTrue:[ Transcript show: ‘Put 1’;cr. anArray at: 1 put: 1.]
```

- 1 m1 will only display the message Put 1 once. Because the array #(nil) is stored into the literal frame of the method and the #at:put: message modified the compiled method itself.

```
SmallInteger>>m2
|anArray|
anArray := Array new: 1.
(anArray at: 1 ) isNil ifTrue:[ Transcript show: ‘Put 1’;cr. anArray at: 1 put: 1]
```

- 1 m2 will always display the message Put 1 because in that case the array is always created at run-time. Therefore it is not detected as a literal at compile-time and not stored into the literal frame of the compiled method. You can find this infomation yourself by defining these methods on a class, inspecting the class and its method dictionary and then the corresponding methods.

Literals, the Arrays (IV)

- This internal representation of method objects has led to the following idioms to prevent unwanted side effects :
  - Never give direct access to a literal array but only provide a copy.

    For example:

```
ar
   ^ #(100@100 200@200) copy
```


Symbols vs. Strings

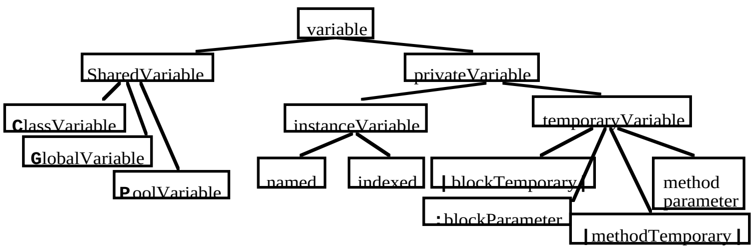
- Symbols are used as method selectors, unique keys for dictionaries
- A symbol is a read-only object, strings are mutable objects
- A symbol is unique, strings are not

```
#calvin == #calvin PrIt-> true
'calvin' == 'calvin' PrIt-> false
#calvin, #zeBest PrIt-> 'calvinzeBest'
```

- Symbols are good candidates for identity based dictionaries (`IdentityDictionary`)
  - Hint: Comparing strings is slower then comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.

Variables Overview

- Maintains a reference to an object
- Dynamically typed and can reference different types of objects
- Shared (starting with uppercase) or private (starting with lowercase)



Temporary Variables

- To hold temporary values during evaluation (method execution)
- Can be accessed by the expressions composing the method body.

```
|mac1 pc node1 printer mac2 packet|
```

- Hint: Avoid using the same name for a temporary variable and a method argument, an instance variable or another temporary variable or block temporary. Your code will be more portable. Do not write:

```
aClass>>printOn: aStream
|aStream|
...
```

- Instead, write:

```
aClass>>printOn: aStream
|anotherStream|
...
```

- Hint: Avoid using the same temporary variable for referencing two different objects

Assignments

- An Assignment is not done by message passing. It is one of the few syntactic elements of Smalltalk.

```
variable := aValue
three := 3 raisedTo: 1
variable1 := variable2 := aValue
```

- In Smalltalk, objects are manipulated via implicit pointers: everything is a pointer. Take care when different variables point to the same object:

```
p1 := p2 := 0@100
p1 x: 100
p1 PrIt-> 100@100
p2 PrIt-> 100@100
```


Method Arguments

- Can be accessed by the expressions composing the method.
- Exist during the execution of the defining method.
- Method Name Example:

```
accept: aPacket
```

In C++ or Java:

```
void Printer::accept(aPacket Packet)
```

- Method arguments cannot change their value within the method body.
  - Invalid Example, assuming `contents` is an instance variable:

```
contents: aString
aString := aString, 'From Lpr'. "concatenates two strings"
addresse := aString
```

  - Valid Example

```
addressee: aString
addressee := aString , 'From Lpr'
```

Instance Variables

- Private to a particular instance (not to all the instances of a class like in C++).
- Can be accessed by all the methods of the defining class and its subclasses.
- Has the same lifetime as the object.

  - Declaration

```
Object subclass: #Node
   instanceVariableNames: 'name nextNode '
...
```

  - Scope

```
Node>>setName: aSymbol nextNode: aNode
   name := aSymbol.
   nextNode := aNode
```

  - But preferably accessed using accessor methods

```
Node>>name
^name
```


Six pseudo-variables

- Smalltalk expressions make references to these variables, but cannot change their values. They are hardwired into the compiler.

1. `nil`
   - nothing, the value for the uninitialized variables. Unique instance of the class `UndefinedObject`
2. `true`
   - unique instance of the class `True`
3. `false`
   - unique instance of the class `False`
   - Hint: Don't use `False` instead of `false`. `false` is the boolean value, `False` the class representing it. So, the first produces an error, the second not:

```
False ifFalse: [Transcript show: 'False']

false ifFalse: [Transcript show: 'False']
```


Six pseudo-variables (II)

- The following variables can only be used in a method body.

4. `self`
   - in the method body it refers to the **receiver** of a message.
5. `super`
   - in the method body it refers also to the **receiver** of the message but its semantics affects the lookup of the method. It starts the lookup in the superclass of the class of the method where the `super` was used and NOT in the superclass of the receiver

```
PrinterServer>>accept: thePacket
"If the packet is addressed to me, print it. Otherwise behave normally."
   (thePacket isAddressedTo: self)
      ifTrue: [self print: thePacket]
      ifFalse: [super accept: thePacket]
```

6. `thisContext`
   - it refers to the instance of `MethodContext` that represents the context of a method (receiver, sender, method, pc, stack). Specific to VisualWorks.

Global Variables

- Always Capitalized
  `MyGlobalPi := 3.1415`
- If it is unknown, Smalltalk will ask you if you want to create a new global
```
Smalltalk at: #MyGlobalPi put: 3.14
MyGlobalPi PrIt-> 3.14
Smalltalk at: #MyGlobalPi PrIt-> 3.14
```
- Stored in the default environment: `Smalltalk` (`aSystemDictionary`)
- Accessible from everywhere, but it is not a good idea to use them; use a classVariable (if shared within an hierarchy or a instance variable of a class) instead
- To remove a global variable:
  `Smalltalk removeKey: #MyGlobal`
- Some predefined global variables:
```
Smalltalk (classes + globals)
Undeclared (aPoolDictionary of undeclared variables accessible from the compiler)
Transcript (System transcript)
ScheduledControllers (window controllers)
Processor (a ProcessScheduler list of all the process)
```


Three Kinds of Messages

- Unary Messages
```
2.4 inspect
macNode name
```
- Binary Messages
```
1 + 2 -> 3
(1 + 2) * (2 + 3) PrIt-> 15
3 * 5 PrIt-> 15
```
- Keyword Messages
```
6 gcd: 24 PrIt-> 6
pcNode nextNode: node2
aLan connectNodesFromAddresses: #(mac node1 pc node2 node3 lpr)
```
- A message is composed of:
  – a receiver, always evaluated `(1+2)`
  – a selector, never evaluated
  – and a list possibly empty of arguments that are all evaluated `(2+3)`
- The receiver is linked with `self` in a method body.

Unary Messages

aReceiver aSelector

```
node3 nextNode -> printerNode
node3 name -> #node3
1 class PrIt-> SmallInteger
false not PrIt-> true
Date today PrIt-> Date today September 19, 1997
Time now PrIt-> 1:22:20 pm
Double pi PrIt-> 3.1415926535898d
```

Binary Messages

aReceiver aSelector anArgument

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

```
1 + 2
2 >= 3
100@100
'the', 'best'
```

- Restriction:
  - second character is never `$-`
  - no mathematical precedence so take care

```
3 + 2 * 10 -> 50
3 + (2 * 10) -> 23
```


Keyword Messages

receiver keyword1: argument1 keyword2: argument2 ...

- In C-like languages it would be:

receiver.keyword1keyword2...(argument1 type1, argument2, type2) : return-type

```
Workstation withName: #Mac2
mac nextNode: node1
Packet send:  'This packet travelled around to the printer' to:  #lw100
aLan createAndDeclareNodesFromAddresses: #(node1 node2 node3) ofKind: Node
1@1 setX: 3
#(1 2 3) at: 2 put: 25
1 to: 10 -> (1 to: 10) anInterval
Browser newOnClass: Point
Interval from:1 to: 20 PrIt-> (1 to: 20)
12 between: 10 and: 20 PrIt-> true
x > 0 ifTrue:['positive'] ifFalse:['negative']
```


Composition

```
69 class inspect
(0@0 extent: 100@100) bottomRight
```

- Precedence Rules:
  - (E) > Unary-E > Binary-E > Keywords-E
  - at same level, from the left to the right

```
2 + 3 squared -> 11
2 raisedTo: 3 + 2 -> 32
#(1 2 3) at: 1+1 put: 10 + 2 * 3 ->  #(1 36 3)
```

- Hint: Use () when two keyword-based messages occur within a single expression, otherwise the precedence order is fine.

```
x isNil ifTrue: [...]
```

- `isNil` is an unary message, so it is evaluated prior to `ifTrue:`

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

- `includes:` is a keyword-based message, it has the same precedence as ifTrue:, so it should be evaluated prior to ifTrue: because the method `includes:ifTrue:` does not exist.

Sequence

message1.
message2.
message3

- . is a separator, not a terminator

```
|macNode pcNode node1 printerNode node2 node3 packet|
"nodes definition"
macNode := Workstation withName: #mac.
pcNode := Workstation withName: #pc.
node1 := Node withName: #node1.
node2 := Node withName: #node2.
node3 := Node withName: #node2.
Transcript cr.
Transcript show: 1 printString.
Transcript cr.
Transcript show: 2 printString
```


Cascade

receiver selector1 [arg] ; selector2 [arg] ; ...

```
Transcript show: 1 printString. Transcript show: cr
```

Is equivalent to:

```
Transcript show: 1 printString ; cr
```

- Important: the semantics of the cascade is to send all the messages in the cascade to the receiver of the FIRST message involved in the cascade.

```
|workst|
workst := Workstation new.
workst name: #mac .
workst nextNode: aNode
```

  - This is equivalent to:

```
Workstation new name: #mac ; nextNode: aNode
```

  - Where `name:` is sent to the newly created instance of workstation and `nextNode:` too.
  - In the following example the FIRST message involved in the cascade is the first `#add:` and not `#with:`. So all the messages will be sent to the result of the parenthesised expression, the newly created instance of `anOrderedCollection`

```
(OrderedCollection with: 1) add: 25; add: 35
```


yourself

- One problem: The following expression returns 35 and not the collection object.

```
(OrderedCollection with: 1) add: 25; add: 35 PrIt-> 35
```

- Let us analyze a bit:

```
OrderedCollection>>add: newObject
"Include newObject as one of the receiver's elements.  Answer newObject."
^self addLast: newObject

OrderedCollection>>addLast: newObject
"Add newObject to the end of the receiver.  Answer newObject."
lastIndex = self basicSize ifTrue: [self makeRoomAtLast].
lastIndex := lastIndex + 1.
self basicAt: lastIndex put: newObject.
^newObject
```

  - How can we reference the receiver of the cascade? By using yourself: `yourself` returns the receiver of the cascade.

```
(OrderedCollection with: 1) add: 25; add: 35 ; yourself
-> OrderedCollection(1 25 35)
```

Did you really understand yourself ?

- yourself returns the receiver of the cascade:

```
Workstation new name: #mac ; nextNode: aNode ; yourself
```

Here the receiver of the cascade is a newly created instance `aWorkstation` and not the class `Workstation.` The `self` in the `yourself` method is linked to this instance (`aWorkstation`)

```
(OrderedCollection with: 1) add: 25; add: 35 ; yourself
anOrderedCollection(1) = self
```

- So if you are that sure that you really understand yourself, what is the code of yourself?

```
Object>>yourself
   ^ self
```


Blocks - Definition

- A deferred sequence of actions
- The Return value is the result of the last expression of the block
- Similar to Lisp Lambda-Expressions, C functions, anonymous functions or procedures

```
[ :variable1 :variable2 |
   | blockTemporary1 blockTemporary2 |
   expression1.
   ...variable1 ...]
```

- Two blocks without arguments and temporary variables

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

- A block with one argument and no temporary variable

```
NetworkManager>>findNodeWithAddress: aSymbol
   "return the first node having the address aSymbol"
   ^self detectNode: [:aNode| aNode name = aSymbol] ifNone: [nil]
```


Blocks - Evaluation

```
[....] value
   or value:
   or value:value:
   or value:value:value:
   or valueWithArguments: anArray
```

- The value of a block is the value of its last statement, except if there is an explicit return ^
- Blocks are first class objects.
- They are created, passed as argument, stored into variables...

fct(x) = x ^ 2 + x

fct (2) = 6

fct (20) = 420

```
|fct|
fct:= [:x | x * x + x].
fct value: 2 PrIt-> 6
fct value: 20 PrIt-> 420
fct PrIt-> aBlockClosure
```

Blocks - Continued

```
|index bloc |
index := 0.
bloc := [index := index +1].
index := 3.
bloc value -> 4

Integer>>factorial
"Answer the factorial of the receiver. Fail if the receiver is less than 0."
| tmp |
....
tmp := 1.
2 to: self do: [:i | tmp := tmp * i].
^tmp
```

- For performance reasons, avoid referring to variables outside a block.

Primitives

- For optimization, if a primitive fails, the code following is executed.

```
Integer>>@ y
"Answer a new Point whose x value is the receiver and whose y value is the argument."

<primitive: 18>
^Point x: self y: y
```

- The End of the Smalltalk World: We need some operations that are not defined as methods on objects but direct calls on the underlying implementation language (C, Assembler,...)

```
== anObject
"Answer true if the receiver and the argument are the same object (have the same
object pointer) and false otherwise.  Do not redefine the message == in any
other class!  No Lookup."

<primitive: 110>
self primitiveFailed

+ - < >* / = == bitShift:\\ bitAnd: bitOr: >= <= at: at:put: new new:
```

7. Dealing with Classes

- Class definition
- Method definition
- Inheritance semantics
- Basic class instantiation

Class Definition: The Class Packet

- A template is proposed by the browser:

```
NameOfSuperclass subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'CategoryName'
```

- Just fill this Template in:

```
Object subclass: #Packet
   instanceVariableNames: 'contents addressee originator '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LAN-Simulation'
```

- Automatically a class named “`Packet class`” is created. `Packet` is the unique instance of `Packet class`. To see it, click on the class button in the browser

Named Instance Variables

```
NameOfSuperclass subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
       ...
Object subclass: #Packet
   instanceVariableNames: 'contents addressee originator '
   ...
```

- Begins with a lowercase letter
- Explicitly declared: a list of instance variables
- Name should be unique because of inheritance
- Default value of instance variable is `nil`
- Private to the instance: instance based (vs. C++ class-based)
- Can be accessed by all the methods of the class and its subclasses (instance methods)
- **Instance variables cannot be accessed by class methods.**
- A client cannot directly access instance variables.
- The clients must invoke accessor methods to access an instance variable.

Method Definition

- Follow the template:

```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

- Fill in the template. For example:

```
Packet>>defaultContents
    “returns the default contents of a Packet”
    ^ ‘contents no specified’

Workstation>>originate: aPacket
    aPacket originator: self.
    self send: aPacket
```

- How to invoke a method on the same object? Send the message to `self`

```
Packet>>isAddressedTo: aNode
    “returns true if I’m addressed to the node aNode”
    ^ self addressee = aNode name
```


Accessing Instance Variables

- Using direct access for the methods of the class

```
Packet>>isSentBy: aNode
    ^ originator = aNode
```

is equivalent to use accessors

```
Packet>>originator
    ^ originator

Packet>>isSentBy: aNode
    ^ self originator = aNode
```

- Get/set accessors for the class Packet:

```
Packet>>addressee
    ^ addressee

Packet>>addressee: aSymbol
    addressee := aSymbol
```

Hint: Do not directly access instance variables of a superclass from the subclass methods. This way classes will not be strongly linked at the structure level.

Methods always return a Value

- Message = effect + return value
- By default, a method returns `self`
- In a method body, the ^ expression returns the value of the expression as the result of the method execution.

```
Node>>accept: thePacket
    "Having received the packet, send it on. This is the default behavior."
    self send: thePacket
```

This is equivalent to:

```
Node>>accept: thePacket
    "Having received the packet, send it on. This is the default behavior."
    self send: thePacket.
    ^self
```

- If we want to return the value returned by `#send:`

```
Node>>accept: thePacket
    "Having received the packet, send it on. This is the default behavior."
    ^self send: thePacket.
```

Some Naming Conventions

- Shared variables begin with an upper case letter
- Private variables begin with a lower case letter
- Use imperative verbs for methods performing an action like `#openOn:`
- For accessors, use the same name as the instance variable accessed:

```
addressee
    ^ addressee

addressee: aSymbol
    addressee := aSymbol
```

- For predicate methods (returning a boolean) prefix the method with `is` or `has`

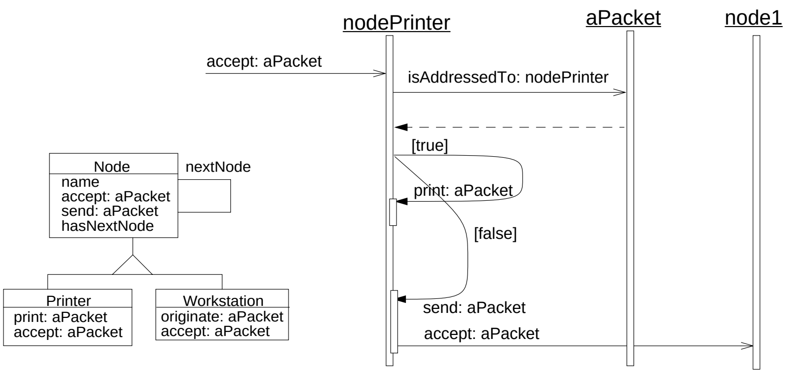
  `isNil, isAddressedTo:, isSentBy:`
- For converting methods prefix the method with `as`

  `asString`

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.

Remember...



Node

```
Object subclass: #Node
   instanceVariableNames: 'name nextNode '
   ...
   Node methodsFor: 'accessing' ....
   Node methodsFor: 'printing' ....
   Node methodsFor: 'send-receive'

   accept: aPacket
   "Having received the packet, send it on. This is the default behavior subclasses
   will probably override me to do something special."

   self hasNextNode ifTrue: [self send: aPacket]

   send: aPacket
   "Precondition: there is a next node. Send a packet to the next node."

   self nextNode accept: aPacket
```


Workstation

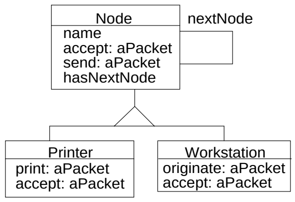
```
Node subclass: #Workstation
   instanceVariableNames: ''
   ...
   Node methodsFor: 'printing' ....
   Node methodsFor: 'send-receive'

   accept: aPacket
   "when a workstation accepts a packet addressed to it, it prints some trace on the transcript"
   (aPacket isAddressedTo: self)
      ifTrue:[ Transcript show: 'A packet  is accepted by the Workstation ', self name asString]
      ifFalse: [super accept: aPacket]

   Node methodsFor: 'send-receive'

   originate: aPacket

   aPacket originator: self.
   self send: aPacket
```


Message Sending & Method Lookup

receiver selector args

- Sending a message is the same as sending a method (associated with the selector and the arguments) to the receiver
- 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.
- The lookup starts in the class of the receiver.
- If the method is defined in the class 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`), a new method called `#doesNotUnderstand:` is sent to the receiver, with a representation of the initial message.

Method Lookup Examples



`node1 accept: aPacket`

7. `node1` is an instance of `Node`
8. `accept:` is looked up in the class `Node`
9. `accept:` is defined in `Node` ⇒ lookup stops + method executed

`macNode accept: aPacket`

1. `macNode` is an instance of `Workstation`
2. `accept:` is looked up in the class `Workstation`
3. `accept:` is defined in `Node` ⇒ lookup stops + method executed

Method Lookup Examples (II)

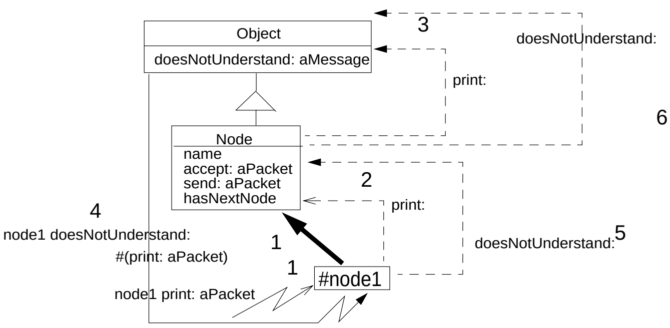
`macNode name`

1. `macNode` is an instance of `Workstation.`
2. `name:` is looked up in the class `Workstation`
3. `name` is not defined in `Workstation` ⇒ lookup continues in `Node`
4. `name` is defined in `Node` ⇒ lookup stops + method executed

`node1 print: aPacket`

1. `node` is an instance of `Node`
2. `print:` is looked up in the class `Node`
3. `print:` is not defined in `Node` ⇒ lookup continues in `Object`
4. `print:` is not defined in `Object` ⇒ lookup stops + exception
5. message: `node1 doesNotUnderstand: #(#print aPacket)` is executed
6. `node1` is an instance of `Node` so `doesNotUnderstand:` is looked up in the class `Node`
7. `doesNotUnderstand:` is not defined in `Node` ⇒ lookup continues in `Object`
8. `doesNotUnderstand:` is defined in `Object` ⇒ lookup stops + method executed (open a dialog box)

Method Lookup Examples (III)



How to Invoke Overridden Methods

- 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
    "when a workstation accepts a packet that is addressed to it,
    it just prints some trace in the transcript"

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

  - 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

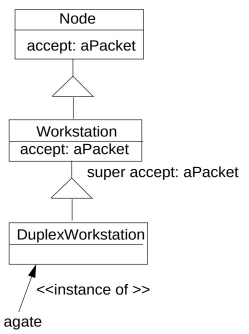
- Like `self`, `super` is a pseudo-variable that refers to the receiver of the message.
- It is used to invoke overridden methods.
- When using `self`, the lookup of the method begins in the **class of the receiver**.
- When using `super,` the lookup of the method begins in the superclass of the class of the method containing the super expression and NOT in the superclass of the receiver class.
  - This means, `super` causes the method lookup to begin searching in the superclass of the class of the method containing `super`

The Semantics of super (II)

- Let us suppose the WRONG hypothesis: "The semantics of super is to start the lookup of a method in the superclass of the receiver class"

`agate accept: aPacket`

1. `agate` is an instance of DuplexWorkstation. `accept:` is looked up in the class DuplexWorkstation
2. `accept:` is not defined in DuplexWorkstation, so the lookup continues in Workstation
3. `accept:` is defined in Workstation, so the lookup stops, and the method is executed
4. `Workstation>>accept:` does a super `accept:`
5. Our hypothesis: super = start the lookup in the superclass of the receiver class. The superclass of the receiver class is `Workstation`



  - This will result in a loop, therefore the hypothesis is WRONG

Object Instantiation

- Objects can be created by:
  - Direct Instance creation:
    `(basic)new/new:`
  - Messages to instances that create other objects
  - Class specific instantiation messages

Instance Creation

- `aClass new/basicNew` returns a newly and UNINITIALIZED instance

```
OrderedCollection new -> OrderedCollection ()
Packet new -> aPacket
Packet new addressee: #mac ; contents: 'hello mac'
```

- Instance variable values = nil
- `#new:/basicNew:` is used to specify the size of the created instance

```
Array new: 4 -> #(nil nil nil nil)
```

- `#new/#new:` can be specialized to define customized creation
- `#basicNew/#basicNew:` should *never* be overridden
- `#new/basicNew` and `new:/basicNew:` are class methods
- Messages to Instances that create Objects

```
1 to: 6                        (an interval)
1@2                            (a point)
(0@0) extent: (100@100)        (a rectangle)
#lulu asString                 (a string)
1 printString                  (a string)
3 asFloat                      (a float)
#(23 2 3 4) asSortedCollection (a sortedCollection)
```


Opening the Box

```
1 to: 6 -> an Interval
Number>>to: stop
   "Answer an Interval from the receiver up to the argument, stop, with
   each next element computed by incrementing the previous one by 1."
   ^Interval from: self to: stop by: 1

1 printString -> aString
Object>>printString
   "Answer a String whose characters are a description of the receiver."
   | aStream |
   aStream := WriteStream on: (String new: 16).
   self printOn: aStream.
   ^aStream contents

1@2 -> aPoint
Number>>@ y
   "Answer a new Point whose x value is the receiver and whose y value is the argument."
   <primitive: 18>
   ^Point x: self y: y
```

Class-specific Instantiation Messages

```
Array with: 1 with: 'lulu'
OrderedCollection with: 1 with:  2 with:  3
Rectangle fromUser -> 179@95 corner: 409@219
Browser browseAllImplementorsOf: #at:put:
Packet send: ‘Hello mac’ to: #mac
Workstation withName: #mac
```

8. Basic Objects, Conditionals and Loops

- Booleans
- Basic Loops
- Overview of the Collection hierarchy— more than 80 classes:
  (`Bag`, `Array`, `OrderedCollection`, `SortedCollection`, `Set`, `Dictionary`...)
- Loops and Iteration abstractions
- Common object behavior

Boolean Objects

- `false` and `true` are objects described by classes `Boolean`, `True` and `False`
- uniform, but optimized and inlined (macro expansion at compile time)
- Logical Comparisons &, |, xor:, not

```
aBooleanExpression comparison anotherBooleanExpression
(1 isZero) & false
```

- Lazy Logical operators

```
aBooleanExpression and: andBlock, aBooleanExpression or: orBlock
```

  `andBlock` will only be valued if `aBooleanExpression` is `true`

  `orBlock` will only be valued if `aBooleanExpression` is `false`

```
false and: [1 error: 'crazy'] PrIt-> false and not an error
```

- Conditionals

```
aBoolean ifTrue: aTrueBlock ifFalse: aFalseBlock
aBoolean ifFalse: aTrueBlock ifTrue: aFalseBlock
aBoolean ifTrue: aTrueBlock
aBoolean ifFalse: aFalseBlock
```

Hint: Take care — `true` is the boolean value and `True` is the class of `true`, its unique instance! Why do conditional expressions use blocks? Because, when a message is sent, the receiver and the arguments of the message are evaluated. Blocks are necessary to avoid evaluating both branches.

Some Basic Loops

```
aBlockTest whileTrue
aBlockTest whileFalse
aBlockTest whileTrue: aBlockBody
aBlockTest whileFalse: aBlockBody
anInteger timesRepeat: aBlockBody

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

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

For the Curious...

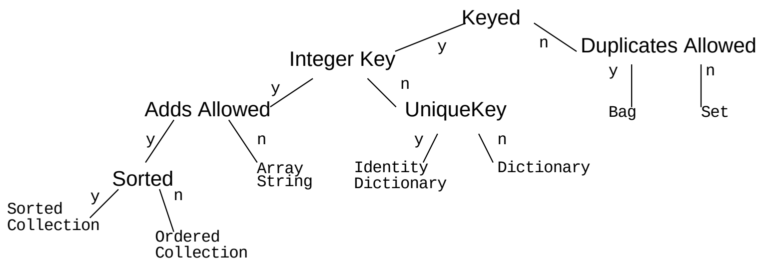
```
BlockClosure>>whileTrue: aBlock
   ^ self value ifTrue: [aBlock value.
                        self whileTrue: aBlock]

BlockClosure>>whileTrue
   ^ [self value] whileTrue:[]

Integer>>timesRepeat: aBlock
"Evaluate the argument, aBlock, the number of times represented by the receiver."

   | count |
   count := 1.
   [count <= self] whileTrue: [aBlock value.
                              count := count + 1]
```


Collections

- Some criteria to identify them
  - Access: indexed, sequential or key-based.
  - Size: fixed or dynamic.
  - Element type: any or well-defined type.
  - Order: defined, defineable or none.
  - Duplicates: possible or not

```
Sequenceable                  ordered
   ArrayedCollection          fixed size + key = integer
      Array                   any kind of elements
      CharacterArray          elements = character
         String
      IntegerArray
   Interval                   arithmetique progression
   LinkedList                 dynamic chaining of the element
   OrderedCollection          size dynamic + arrival order
      SortedCollection        explicit order
Bag                           possible duplicate + no order
Set                           no duplicate + no order
   IdentitySet                identification based on identity
   Dictionary                 element = associations + key based
      IdentityDictionary      key based on identity
```


Collections - Another View



Collection Methods

- Will be **defined, redefined, optimized or forbidden** in the subclasses
  - Accessing: `#size, #capacity, #at: anInteger, #at: anInteger put: anElement`
  - Testing: `#isEmpty, #includes: anElement, #contains: aBlock, occurencesOf: anElement`
  - Adding: `#add: anElement, #addAll: aCollection`
  - Removing: `#remove: anElement, #remove:anElement ifAbsent: aBlock, #removeAll: aCollection`
  - Enumerating (See generic enumerating): `#do: aBlock, #collect: aBlock, #select: aBlock, #reject: aBlock, #detect:, #detect: aBlock ifNone: aNoneBlock, #inject: avalue into: aBinaryBlock`
  - Converting: `#asBag, #asSet, #asOrderedCollection, #asSortedCollection, #asArray, #asSortedCollection: aBlock`
  - Creation: `#with: anElement, #with:with:, #with:with:with:, #with:with:with:with:, #with:All: aCollection`

Sequenceable Specific (Array)

```
|arr|
arr := #(calvin hates suzie).
arr at: 2 put: #loves.
arr PrIt-> #(#calvin #loves #suzie)
```

- Accessing: `#first, #last, #atAllPut: anElement, #atAll: anIndexCollection: put: anElement`
- Searching (*: + ifAbsent:): `#indexOf: anElement, #indexOf: anElement ifAbsent: aBlock`
- Changing: `#replaceAll: anElement with: anotherElement`
- Copying: `#copyFrom: first to: last, copyWith: anElement, copyWithout: anElement`

KeyedCollection Specific (Dictionary)

```
|dict|
dict := Dictionary new.
dict at: 'toto' put: 3.
dict at: 'titi' ifAbsent: [4]. -> 4
dict at: 'titi' put: 5.
dict removeKey: 'toto'.
dict keys -> Set ('titi')
```

- Accessing: `#at: aKey, #at: aKey ifAbsent: aBlock, #at: aKey ifAbsentPut: aBlock, #at: aKey put: aValue, #keys, #values, #associations`
- Removing: `#removeKey: aKey, #removeKey: aKey ifAbsent: aBlock`
- Testing: `#includeKey: aKey`
- Enumerating: `#keysAndValuesDo: aBlock, #associationsDo: aBlock, #keysDo: aBlock`

Choose your Camp!

- You could write:

```
absolute: aCollection
   |result|
   result := aCollection species new: aCollection size.
   1 to: aCollection size do:
      [ :each | result at: each put: (aCollection at: each) abs].
   ^ result
```

- You could also write:

```
absolute: aCollection
   ^ aCollection collect: [:each| each abs]
```

Really important: Contrary to the first solution, the second solution works well for indexable collections and also for sets.

Iteration Abstraction: do:/collect:

```
aCollection do: aOneParameterBlock
aCollection collect: aOneParameterBlock
aCollection with: anotherCollection do: aBinaryBlock
```

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

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

#(1 2 3) with: #(10 20 30)
         do: [:x :y| Transcript show: (y ** x) printString ; cr ]
```


Iteration Abstraction: select:/reject:/detect:

```
aCollection select: aPredicateBlock
aCollection reject: aPredicateBlock
aCollection detect: aOneParameterPredicateBlock
aCollection
          detect: aOneParameterPredicateBlock
          ifNone: aNoneBlock
```

```
#(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] PrIt-> 19
#(12 10 12 68) detect: [:i|i odd] ifNone:[1] PrIt-> 1
```

Iteration Abstraction: inject:into:

```
aCollection inject: aStartValue into: aBinaryBlock
```

```
|acc|
acc := 0.
#(1 2 3 4 5) do: [:element | acc := acc + element].
acc
-> 15
```

```
#(1 2 3 4 5)
   inject: 0
   into: [:acc :element| acc + element]
-> 15
```


Collection Abstraction

```
aCollection includes: anElement
aCollection size
aCollection isEmpty
aCollection contains: aBooleanBlock
```

```
#(1 2 3 4 5) includes: 4 -> true
#(1 2 3 4 5) size -> 5
#(1 2 3 4 5) isEmpty -> false
#(1 2 3 4 5) contains: [:each | each isOdd] -> true
```


Examples of Use: NetworkManager

```
aLan findNodeWithAddress: #mac
```

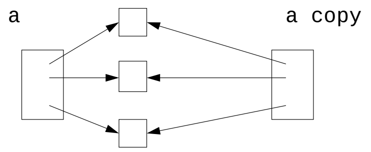
```
NetworkManager>>findNodeWithAddress: aSymbol

   ^self findNodeWithAddress: aSymbol ifNone: [nil]

NetworkManager>>findNodeWithAddress: aSymbol ifNone: aBlock

   ^nodes detect: [:aNode| aNode name = aSymbol] ifNone: aBlock
```

```
aLan createAndDeclareNodesFromAddresses: #(node1 node2 node3) ofKind: Node
```

```
NetworkManager>>createAndDeclareNodesFromAddresses: anArrayOfAddresses ofKind: aNodeClass
   "given a list of addresses, create the corresponding nodes of the aNodeClass kind"

   (Node withAllSubclasses includes: aNodeClass)
      ifTrue: [anArrayOfAddresses do: [:each | self declareNode: (aNodeClass withName: each)]]
      ifFalse: [self error: aNodeClass name , ' is not a class of nodes']
```

Common Shared Behavior

- Object is the root of the inheritance tree
- Defines the common and minimal behavior for all the objects in the system. It has 161 instance methods and 19 class methods
- Comparison of objects: `#==, #~~, #=, #=~, #isNil, #notNil`
- Copying of objects: `#shallowCopy, #copy`
  `#shallowCopy` : the copy shares instance variables with the receiver.
  default implementation of `#copy` is `#shallowCopy`



Identity vs. Equality

- `= anObject` returns `true` if the structures are equivalent (the same hash number)

```
(Array with: 1 with: 2) = (Array with:1 with:2) PrIt-> true
```

- `== anObject` returns true if the receiver and the argument point to the same object. #== should never be overridden. On Object #= is #==.
- `~= is not =`
- `~~` is `not ==`

```
(Array with: 1 with: 2 ) == (Array with: 1 with:2) PrIt-> false
(Array with: 1 with: 2 ) = (Array with: 1 with:2) PrIt-> true
```

- Take care when redefining #= . One should override `#hash` too!

Common Shared Behavior (II)

- Print and store objects: `#printString, #printOn: aStream, #storeString, #storeOn: aStream`

```
#(123 1 2 3) printString -> '#(123 1 2 3)'
Date today printString -> 'October 5, 1997'
Date today storeString -> '(Date readFromString: ''10/5/1997'')'
OrderedCollection new add: 4 ; add: 3 ; storeString ->
'((OrderedCollection new) add: 4; add: 3; yourself)'
```

- You need the compiler, so for a deployment image this is not convenient
- Create instances from stored objects: class methods `readFrom: aStream, readFromString: aString`

```
Object readFromString:  '((OrderedCollection new) add: 4; yourself)'
-> OrderedCollection (4)
```

- Notifying the programmer:

```
#error: aString, #doesNotUnderstand: aMessage,
#halt, #shouldNotImplement, #subclassResponsibility
```

Common Shared Behavior (III)

- `#class` returns the class of the object
- `#inspect` opens an inspector
- `#browse` opens a browser
- `#halt` stops the execution and opens a debugger (to be inserted in a body of a method)
- `#printString` (calls `#printOn:`) returns a string representing the object
- `#storeString` returns a string whose evaluation recreates an object equal to the receiver
- `#readFromString: aStream` recreates an object

9. Numbers

The Basics of Numbers

- Arithmetic
  `5 + 6, 5 - 6, 5 * 6,`
  division `30 / 9`, integer division `30 // 9` , modulo `30 \\ 9`
  square root `9 sqrt`, square `3 squared`
- Rounding
  `3.8 ceiling -> 4`
  `3.8 floor -> 3`
  `3.811 roundTo: 0.01 -> 3.81`
- Range
  `30 between: 5 and: 40`
- Tests
  `3.8 isInteger`
  `3.8 even, 3.8 odd`
- Signs
  `positive, negative, sign, negated`
- Other
  `min:, max:, cos, ln, log, log: arcSin, exp, **`

Deeper into Numbers: Double Dispatch

- How to select a method depending on the receiver AND the argument?
  - Send a message back to the argument *passing the receiver as an argument*
- Example: Coercion between Float and Integer
  - A not very good solution:

```
Integer>>+ aNumber
    (aNumber isKindOf: Float)
        ifTrue: [ aNumber asFloat + self]
        ifFalse: [ self addPrimitive: aNumber]

Float>>+ aNumber
    (aNumber isKindOf: Integer)
        ifTrue: [aNumber asFloat + self]
        ifFalse: [self addPrimitive: aNumber]
```

Deeper into Numbers: Double Dispatch (II)

```
(a)    Integer>>+ aNumber
           ^ aNumber sumFromInteger: self
(b)    Float>>+ aNumber
           ^ aNumber sumFromFloat: self
```

```
(c) Integer>>sumFromInteger: anInteger
          <primitive: 40>
(d) Float>>sumFromInteger: anInteger
          ^ anInteger asFloat + self

(e) Integer>>sumFromFloat: aFloat
          ^aFloat + self asFloat
(f) Float>>sumFromFloat: aFloat
          <primitive: 41>
```

- Some Tests:

```
1 + 1: (a->c)
1.0 + 1.0: (b->f)
1 + 1.0: (a->d->b->f)
1.0 + 1: (b->e->b->f)
```


Deeper into Numbers: Coercion & Generality

```
ArithmeticValue>>coerce: aNumber
   "Answer a number representing the argument, aNumber, that is the same kind of Number
   as the receiver.  Must be defined by all Number classes."

   ^self subclassResponsibility

ArithmicValue>>generality
   "Answer the number representing the ordering of the receiver in the generality hierarchy.  A
number
   in this hierarchy coerces to numbers higher  in  hierarchy  (i.e.,  with  larger  generality  num-
bers)."

   ^self subclassResponsibility

Integer>>coerce: aNumber
   "Convert a number to a compatible form"

   ^aNumber asInteger

Integer>>generality
   ^40

Generality
   SmallInteger 20
   Integer 40
   Fraction 60
   FixedPoint 70
   Float 80
   Double 90
```


Deeper into Numbers: #retry:coercing:

```
ArithmeticValue>>sumFromInteger: anInteger
   "The argument anInteger, known to be a kind of integer,
   encountered a problem on addition. Retry by coercing either
   anInteger or self, whichever is the less general arithmetic value."
   Transcript show: 'here arthmeticValue>>sunFromInteger' ;cr.
   ^anInteger retry: #+ coercing: self


ArithmeticValue>>retry: aSymbol coercing: aNumber
   "Arithmetic represented by the symbol, aSymbol, could not be performed with the receiver and the
   argument, aNumber, because of the differences in representation.  Coerce either the receiver or
   the argument, depending on which has higher generality, and try again. If the generalities are
the
   same, then this message should not have been sent so an error notification is provided."

   self generality < aNumber generality
      ifTrue: [^(aNumber coerce: self) perform: aSymbol with: aNumber].
   self generality > aNumber generality
      ifTrue: [^self perform: aSymbol with: (self coerce: aNumber)].
   self error: 'coercion attempt failed'
```

10. Exceptions

- Standardized by ANSI and available since VW 3.0
- `Exception` is the root of the exception hierarchy: 84 predefined exceptions. The two most important classes are:
  - Error
  - Notification
- Specialised into predefined exceptions -> subclass them to create your own exceptions
- Some methods of Exception:
  - `defaultAction` is executed when an exception occurs
  - `description` string describing the actual exception

The Main Exceptions

Exception class	Exceptional Event	Default Action
Error	Any program error	Open a Notifier
ArithmeticError	Any error evaluating an arithmetic	Inherited from Error
MessageNotUnderstood	A message was sent to an object that did not define a corresponding method	Inherited from Error
Notification	Any unusual event that does not impair continued execution of the program	Do nothing continuing executing
Warning	An unusual event that the user should be informed about	Display Yes/No dialog and return a boolean value t the signaler
ZeroDivide		Inherited from ArithmeticError

Basic Example of Catching

```
|x y|
x := 7. y := 0.
[x/y]
   on: ZeroDivide
   do: [:exception| Transcript show: exception description, cr.
                   0....]
```

- an Exception Handler is defined using `on:do:` and is composed by an exception class and a handler block

```
ZeroDivide
   [:theException| Transcript show: ‘ division by zero’]
```

- An Exception Handler completes by returning the value of the handler block in place of the value of the protected block (here [x/y]).
- We can exit the current method by putting an explicit return inside the handler block

Exception Sets

- Exception Sets
```
[do some work]
   on: ZeroDivide, Warning
   do: [ :ex| what you want]
```
  Or
```
|exceptionSets|
exceptionSets := ExceptionSet with: ZeroDivide with: Warning.
[do some work]
   on: exceptionSets
   do: [ :ex| what you want]
```
- Signaling an Exception:
```
Error raiseSignal
Warning raiseSignal: 'description of the exception'
```


Exception Environment

- Each process has its own exception environment: an ordered list of active handlers.
  - Process starts –> list empty
  - [aaaa] on: Error do: [bbb] –> Error,bbb added to the beginning of the list
  - When an exception is signaled, the system sends a message to the first handler of the exception handler.
  - If the handler cannot handle the exception, the next one is asked
  - If no handler can handle the exception then the default action is performed

Resumable and Non-Resumable

- A handler block completes by executing the last statement of the block. The value of the last statement is then the value returned by the handler block. Where this value should be returned depends:
  - Nonresumable: like Error -> 'Value from handler'
```
([Error raiseSignal. 'Value from protected block']
   on: Error
   do: [:ex|ex return: 'Value from handler'])
```
  - Resumable: like Warning, Notification -> 'Value from protected Block'. In this case Notification raiseSignal raises an exception, then the context is restored and the value returned normally
```
([Notification raiseSignal. 'Value from protected block']
   on: Notification
   do: [:ex|ex resume: 'Value from handler'])
```

Resume:/Return:

```
Transcript show:
      ([Notification raiseSignal. 'Value from protected block']
         on: Notification
         do: [:ex| Transcript show: 'Entering handler '.
         'Value from handler'. '5'])
-> Entering handler 5
Transcript show:
      ([Notification raiseSignal. 'Value from protected block']
         on: Notification
         do: [:ex| Transcript show: 'Entering handler '.
         ex resume: 'Value from handler'. '5'])
-> Entering handler Value from protected block
Transcript show:
      ([Notification raiseSignal. 'Value from protected block']
         on: Notification
         do: [:ex| Transcript show: 'Entering handler '.
         ex return: 'Value from handler'. '5'])
-> Entering handler Value from handler
```


Exiting Handlers Explicitly

- `exit` or `exit:` (VW specific) Resumes on a resumable and returns on a nonresumable exception
- `resume` or `resume:` Attempts to continue processing the protected block, immeditely following the message that triggered the exception.
- `return` or `return:` ends processing the protected block that triggered the exception
- `retry` re-evaluates the protected block
- `retryUsing:` evaluates a new block in place of the protected block
- `resignalAs:` resignal the exception as another one
- `pass` exit the current handler and pass to the next outer handler, control does not return to the passer
- `outer` as with pass, except will regain control if the outer handler resumes

exit:, resume: and return: return their argument as the return value, instead of the value of the final statement of the handler block

Examples

- Look in Exception class examples categories

```
-2.0 to: 2.0 do:
[ :i |
   [ 10.0 / i. Transcript cr; show: i printString ]
      on: Number divisionByZeroSignal do:
      [:ex | Transcript cr; show: 'divideByZero abort'.
      ex return ]
]

-2.0
-1.0
divideByZero abort
1.0
2.0
```

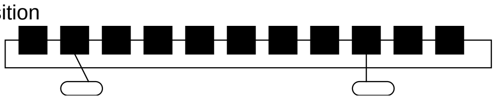
- `retry` recreates the exception environment of active handlers

```
[ x /y]
   on: ZeroDivide
   do: [:exception|
   y := 0.00001.
   exception retry]
```

11. Streams

Streams

- Allows the traversal of a collection
- Associated with a collection
  - If the collection is a Smalltalk collection: `InternalStream`
  - If the collection is a file or an object that behaves like a collection: `ExternalStream`
- Stores the current position



```
Stream (abstract)
        PeekableStream (abstract)
            PositionableStream (abstract)
                ExternalStream
                        ExternalReadStream
                            ExternalReadAppendStream
                            ExternalReadWriteStream
                        ExternalWriteStream
                InternalStream
                    ReadStream
                    WriteStream
                        ReadWriteStream
```


An Example

```
|st|
st := ReadWriteStream on: (OrderedCollection new: 5).
st nextPut: 1.
st nextPutAll: #(4 8 2 6 7).
st contents. PrIt-> OrderedCollection (1 4 8 2 6 7)
st reset.
st next. -> 1
st position: 3.
st next. -> 2
st := #(1 2 5 3 7) readStream.
st next. -> 1
```

printString, printOn:

```
Object>>printString
   "Answer a String whose characters are a description of the receiver."

   | aStream |
   aStream := WriteStream on: (String new: 16).
   self printOn: aStream.
   ^aStream contents

Node>>printOn: aStream

   super printOn: aStream.
   aStream nextPutAll: ' with name:'; print: self name.
   self hasNextNode  ifTrue: [
      aStream nextPutAll: ' and next node:'; print: self nextNode name]
```


Stream Classes

- Stream
  - `#next` returns the next element
  - `#next: n` returns the n next elements
  - `#contents` returns all the elements
  - `#nextPut: anElement` inserts `anElement` at the next position
  - `#nextPutAll: aCollection` inserts the collection element from the next position on
  - `#atEnd` returns true if at the end of the collection
- PeekableStream: Access to the current without passing to the next
  - `#peek`
  - `#skipFor: anArgument`
  - `#skip: n` increases the position of n
  - `#skipUpTo: anElement` increases the position after anElement
  - `#on: aCollection,` creates a stream
  - `#on: aCol from: firstIndex to: lastIndex` (index elements included)

Stream Classes (II)

- PositionableStream
  - `#skipToAll: #throughAll: #upToAll:`
  - `#position`
  - `#position: anInteger`
  - `#reset #setToEnd #isEmpty`
- InternalStream
  - `#size` returns the size of the internal collection
  - Creation `#with:` (without reinitializing the stream)
- ReadStream
- WriteStream
- ReadWriteStream
- ExternalStream and subclasses

Stream Tricks

- `Transcript` is a `TextCollector` that has aStream

```
TextCollector>>show: aString
   self nextPutAll: aString.
   self endEntry
```

- `#endEntry` via dependencies asks for refreshing the window. If you want to speed up a slow trace, use `#nextPutAll:` + `#endEntry` instead of `#show:`

```
|st sc|
st := ReadStream on: ‘we are the champions’.
sc := Scanner new on: st.
[st atEnd] whileFalse: [ Transcript nextPutAll: sc scanToken, ‘ * ‘].
Transcript endEntry
```


Streams, Blocks and Files

- How to ensure that the open files are closed

```
MyClass>readFile: aFilename
|readStream|
readStream := aFilename readStream.
[[readStream atEnd] whileFalse: [....]]
   valueNowOrOnUnwindDo: [readStream close]
```

- How to find open files (VW specific)

```
(ExternalStream classPool at: #OpenStreams) copy inspect
```

- Filename

```
#appendStream (addition + creation if file doesnot exists)
#newReadAppendStream, #newReadWriteStream (if receiver exists, contents removed)
#readAppendStream, #readWriteStream, #readStream, #writeStream
```

- Example: Removing Smalltalk comments from a file

```
|inStream outStream |
inStream := (Filename named: ‘/home/ducasse/test.st’) readStream.
outStream := (Filename named: ‘/home/ducasse/testout.st’) writeStream.
“(or ‘/home/ducasse/ducasse’ asFilename)”
[inStream atEnd] whileFalse: [
   outStream nextPutAll: (inStream upTo: $”).
   inStream skipTo: $”].
   ^outStream contents     “do not forget to close the files too
```

Part II - Advanced Smalltalk Elements

- Advanced Classes
- The Model-View-Controller Paradigm
- Processes and Concurrency
- Classes and Metaclasses - an Analysis
- Common Mistakes and Debugging
- The Internal Structure of Objects
- Blocks and Optimization
- Advanced Blocks

12. Advanced Classes

- Indexed Classes
- Classes as Objects
- Class Instance Variables and Methods
- Class Variables
- Pool Dictionaries

Types of Classes

Indexed	Named	Definition Method	Examples
No	Yes	`#subclass:...`	`Packet, Workstation`
Yes	Yes	`#variableSubclass:`	`Array, CompiledMethod`
Yes	No	`#variableByteSubclass`	`String, ByteArray`

- Method related to class types: `#isPointers, #isBits, #isBytes, #isFixed, #isVariable, #kindOfSubclass`
- classes defined using `#subclass:` support any kind of subclasses
- classes defined using `#variableSubclass:` can only have: `variableSubclass:` or `variableByteSubclass:`subclasses
- classes defined using `#variableByteSubclass`
  - can only be defined if the superclass has no defined instance variable
  - pointer classes and byte classes don't mix
  - only byte subclasses

Two Views on Classes

- Named or indexed instance variables
  Named: `'addressee'` of `Packet`
  Indexed: `Array`

- Or looking at them in another way:
  Objects with pointers to other objects
  Objects with arrays of bytes (word, long)

  Difference for efficiency reasons: arrays of bytes (like C strings) are faster than storing an array of pointers, each pointing to a single byte.

Indexed Classes

- For classes that need a variable number of instance variables

  Example: the class `Array`

```
ArrayedCollection variableSubclass: #Array
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
    category: 'Collections-Arrayed'

Array new: 4 -> #(nil nil nil nil)
#(1 2 3 4) class isVariable -> true
```


Indexed Classes / Instance Variables

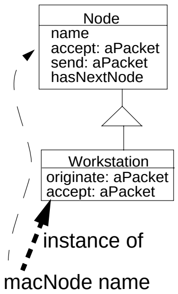
- Indexed variable is implictly added to the list of instance variables
- Only one indexed instance variable per class
- Access with `#at:` and `#at:put:`
  (`#at:put:` answers the value, not the receiver)
- First access: `anInstance at: 1`
- `#size` returns the number of indexed instance variables
- Instantiated with `#new:` max

```
|t|
t := (Array new: 4).
t at: 2 put: 'lulu'.
t at: 1 -> nil
```

- Subclasses should also be indexed

The meaning of “Instance of”

- Every object is an instance of a class.
- Every class is ultimately a subclass of `Object` (except `Object`).
- When anObject receives a message, the method is looked up in its class and/or its superclasses.
- A class defines the structure and the behavior of all its instances.
- Each instance possesses its own set of values.
- Each instance shares the behavior defined in its class with other instances via the **instance of** link.



Example:
`macNode name`

1. `macNode` is an instance of `Workstation` ⇒ `name` is looked up in the class `Workstation`
2. `name` is not defined in Workstation ⇒ lookup continues in `Node`
3. `name` is defined in `Node` ⇒ lookup stops + method executed

The meaning of “Instance of” (II)

- A class is an object too, so messages sent to it are looked up into the class of the class, its metaclass.
- Every class (X) is the unique instance of its associated metaclass named X class

Example:

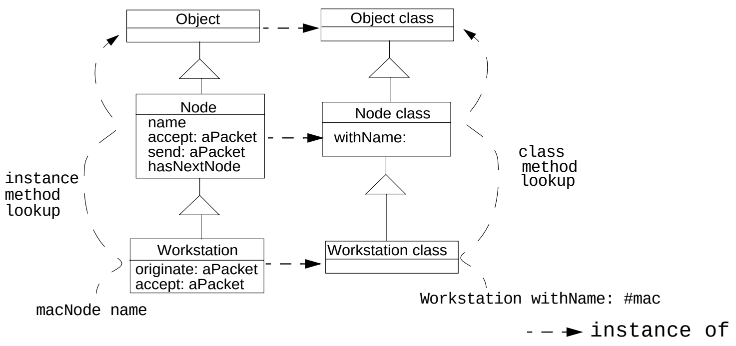
`Node withName: #node1`

1. `Node` is an instance of `Node class` ⇒ `withName:` is looked up in the class `Node class`
2. `withName:` defined in `Node class` ⇒ lookup stops + method executed

`Workstation withName: #mac`

1. `Workstation` is an instance of `Workstation class` ⇒ `withName:` is looked up in the class `Workstation class`
2. `withName:` is not defined in `Workstation class` ⇒ lookup continues in the superclass of `Workstation` class = `Node class`
3. `withName:` is defined in `Node class` ⇒ lookup stops + method executed

Lookup and Class Messages



The Meaning of “Instance of” (III)

`Node new: #node1`

1. `Node` is an instance of `Node class` ⇒ `new:` is looked up in the class `Node class`
2. `new:` is not defined in `Node class` ⇒ lookup continues in the superclass of `Node class` = `Object class`
3. `new:` is not defined in `Object class` ⇒ lookup continues in the superclass of `Node class` ....`Class`, `ClassDescription`, `Behavior`
4. `new:` is defined in `Behavior` ⇒ lookup stops + method executed.

- This is the same for Array new: 4
  new: is defined in Behavior (the ancestor of Array class)

  Hint: Behavior is the essence of a class. ClassDescription represents the extra functionality for browsing the class. Class supports poolVariable and classVariable.

Metaclass Concepts & Responsibilities

- Concepts:
  - Everything is an object
  - Each object is instance of one class
  - A class (X) is also an object, the **sole** instance of its associated metaclass named X class
  - An object is a class if and only if it can create instances of itself.
- Metaclass Responsibilities:
  - instance creation
  - class information (inheritance link, instance variables, method compilation...)
- Examples:

```
Node allSubclasses -> OrderedCollection (WorkStation OutputServer Workstation File
   Server PrintServer)
LanPrint allInstances -> #()
Node instVarNames -> #('name' 'nextNode')
Workstation withName: #mac -> aWorkstation
Workstation selectors  -> IdentitySet (#accept: #originate:)
Workstation canUnderstand: #nextNode -> true
```


Class Instance Variables

- Like any object, a class is an instance of a class that can have instance variables that represent the state of a class.
- Singleton Design Pattern: a class with only one instance

```
NetworkManager class
   instanceVariableNames: 'uniqueInstance'
```

- `NetworkManager` being an instance of `NetworkManager class` has an instance variable named `uniqueInstance.`
  - Hint: An instance variable of a class can be used to represent information shared by all the instances of the class. However, you should use class instance variables to represent the state of the class (like the number of instances, ...) and use classVariable instead.

About Behavior

- `Behavior` is the first metaclass. All other metaclasses inherit from it
- `Behavior` describes the minimal structure of a class:
  - superclass and subclasses
  - method dictionary
  - format (instance variable compressed description)

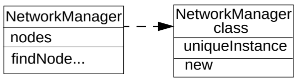
```
Object subclass: #Behavior
   instanceVariableNames: 'superclass methodDict format subclasses '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Classes'
```

- Example of Queries

```
Packet superclass -> Object
Packet subclasses - #()
Packet selectors -> IdentitySet (#originator: #addressee: #addressee
   #isOriginatedFrom: #printOn: #isAddressedTo: #originator #initialize
   #contents #contents:)
Packet allInstVarNames -> OrderedCollection ('addressee' 'originator'
   'contents' 'visitedNodes')
Packet isDirectSubclassOf: Object -> true
```

Class Method

- As any object a metaclass can have methods that represent the behavior of a class.
- Some examples of class behavior:
  - class definition, finding all instances of a class
  - navigation in the hierarchy,
  - finding the instance variable names, methods
  - instance creation, compiling methods



- Can only access instance variable of the class:
  Examples: `NetworkManager class>>new` can only access `uniqueInstance` class instance variable and not instance variables (like `nodes`).
- Default Instance Creation class method:
  - new/new: and basicNew/basicNew: (see Direct Instance Creation)

```
Packet new
```

- Specific instance creation method

```
Packet send: 'Smalltalk is fun' to: #lpr
Workstation withName: #mac
Workstation withName: #mac connectedTo: #lpr
```


classVariable

- How to share state between all the instances of a class: Use classVariable
  - a classVariable is **shared** and directly accessible by all the instances of the class and subclasses
  - A pretty bad name: should have been called **Shared Variables**
  - Shared Variable ⇒ begins with an uppercase letter
  - a classVariable can be directly accessed in instance methods and class methods

```
NameOfSuperclass subclass: #NameOfClass
    ...
    classVariableNames: 'ClassVarName1 ClassVarName2'
    ...
Object subclass: #NetworkManager
    ...
    classVariableNames: 'Domain'
```

- Sometimes classVariable can be replaced by class methods

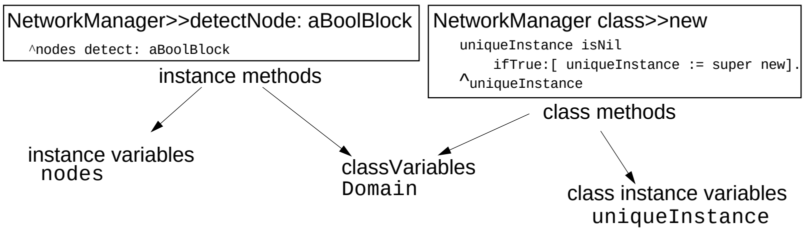
```
NetworkManager class>>domain
    ^ 'iam.unibe.ch'
```


Class Instance Variables / Class Variables

- a classVariable is **shared** and directly accessible by all the instances and subclasses
- Class instance variables, just like normal instance variables, can be accessed only via class message and accessors:
  - an instance variable of a class is private to this class.
  - an instance
- Take care: when you change the value of a classVariable the whole inheritance tree is impacted!
- ClassVariables can be used in conjunction with instance variables to cache some common values that can be changed locally in the classes.
- Examples: in the `Scanner` class a table describes the types of the characters (strings, comments, binary....). The original table is stored into a classVariable, its value is loaded into the instance variable. It is then possible to change the value of the instance variable to have a different scanner.

```
Object subclass: #Scanner
    instanceVariableNames: 'source mark prevEnd hereChar token tokenType buffer typeTable '
    classVariableNames: 'TypeTable '
    category: 'System-Compiler-Public Access'
```

Summary of Variable Visibility



Example From The System: Geometric Class

```
Object subclass: #Geometric
    instanceVariableNames: ''
    classVariableNames: 'InverseScale Scale '
    ...

Geometric class>>initialize
    "Reset the class variables."

    Scale := 4096.
    InverseScale := 1.0 / Scale
```


Circle

```
Geometric subclass: #Circle
    instanceVariableNames: 'center radius'
    classVariableNames: ''

Circle>>center
    ^center

Circle>>setCenter: aPoint radius: aNumber
    center := aPoint.
    radius := aNumber

Circle>>area
    | r |
    r := self radius asLimitedPrecisionReal.
    ^r class pi * r * r

Circle>>diameter
    ^self radius * 2

Circle class>>center: aPoint radius: aNumber
    ^self basicNew setCenter: aPoint radius: aNumber
```

poolDictionaries

- Also called Pool Variables.
- Shared variable ⇒ begins with a uppercase letter.
- Variable shared by a group of classes not linked by inheritance.
- Each class possesses its own pool dictionary.
- They are not inherited.

- Examples of PoolDictionaries from the System:Text

```
CharacterArray subclass: #Text
      instanceVariableNames: 'string runs '
      classVariableNames: ''
      poolDictionaries: 'TextConstants '
      category: 'Collections-Text'
```

- Elements stored into TextConstants like Ctrl, CR, ESC, Space can be directly accessed from all the classes like ParagraphEditor....
- On VW poolDictionary should not be an IdentityDictionary

Example of PoolVariables

- Instead of

```
Smalltalk at: #NetworkConstant put: Dictionary new.
NetworkConstant at: #rates put: 9000.
Node>>computeAverageSpeed
   ...
NetworkConstant at: #rates
```

Write:

```
Object subclass: #Packet
   instanceVariableNames: 'contents addressee originator '
   classVariableNames: ‘Domain’
   poolDictionaries: 'NetworkConstant'

Node>>computeAverageSpeed
...
   .. rates
```

- `rates` is directly accessed in the **global** dictionary `NetworkConstant`.
- As a beginner policy, do not use `poolDictionaries`

13. The Model-View-Controller Paradigm

- Commonly named MVC
- Not a tutorial on how to build user interface (look at the exercises)
- => Observer pattern in Smalltalk

Context

- Building interactive applications with a Graphical User Interface
  - Obvious example: the Smalltalk Development Environment
- Characteristics of such applications:
  - Event driven user interaction, not predictable -> Interface Code can get very complex
  - Interfaces are often subject of changes.

“As far as the user is concerned, the interface IS the program.”

- Question: How can we reduce the complexity of developing such applications?
- The Answer is Modularity

Program Architecture

- A **Software Architecture** is a collection of software and system components, connections between them and a number of constraints they have to fulfill.
- Goals we want to achieve with our architecture:
  - manageable complexity
  - reusability of the individual components
  - pluggability, i.e., an easy realization of the connections between the components
- The Solution for the domain of GUI-driven applications is to partition an application as follows:
  - Model
  - View
  - Controller
  - This leads to a separation of concerns

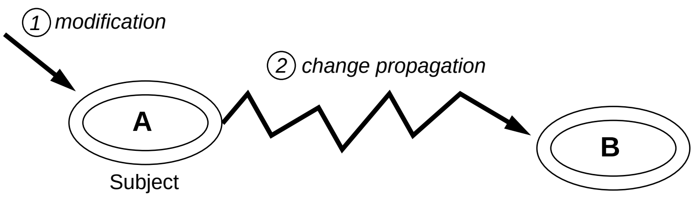
Separation of Concerns

Functionality vs. Display and Display vs. Interaction

- Model is the Functionality, i.e., the Domain
  - Domain specific information
  - Core functionality, where the computation/data processing takes place
- View is the Display, i.e., the User Interface
  - Presentation of the data in various formats
  - “What the user sees”
  - dealing with user input (Mouse, Keyboard, etc.)
- Controller is the Interaction
  - relaying the user input to the View (e.g., scrolling, resizing) or the model (e.g., modification of the data)
- View and Controller are very much related. There is always a 1:1 relationship between views and controllers. There are also examples of systems where view and controller are not separated. Rationale for separating View and Controller:
  - – reusability of the individual components and freedom of choice is better:
    the same view with different controllers (different modes of interaction)
    the same controller for different views (Action Button/Radio Button)

The notion of Dependency

- An object B that **depends on** another object A must be informed about changes in the state of A, in order to be able to adapt its own state.

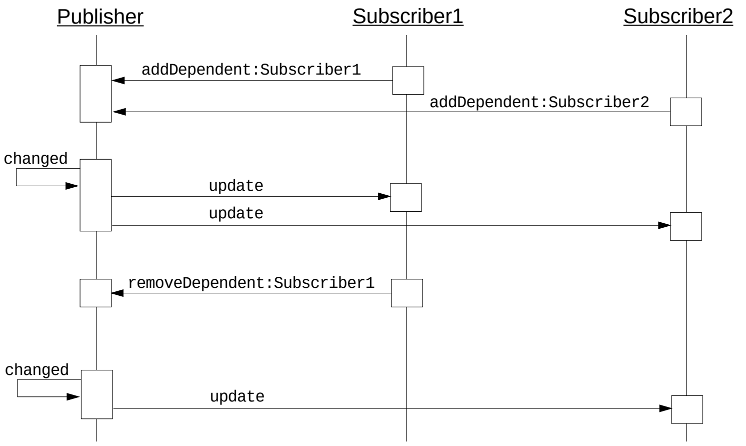


- Dependencies that are realised via messages sent directly to dependent objects are not very reusable and are likely to break in times of change.
  - Decoupling of subject and dependent

Dependency Mechanism

- The Publisher-Subscriber Pattern (a.k.a. Observer Pattern)
- Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The pattern ensures the automatisation of
  - adding and removing dependents
  - change propagation
- The publisher (subject) has a list of subscribers (observers, dependents). A subscriber registers with a publisher.

Publisher-Subscriber: A Sample Session



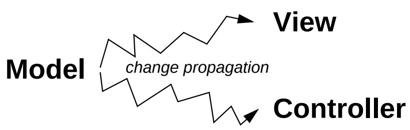
Change Propagation: Push and Pull

How is the changed data transferred
from the publisher to the subscriber?

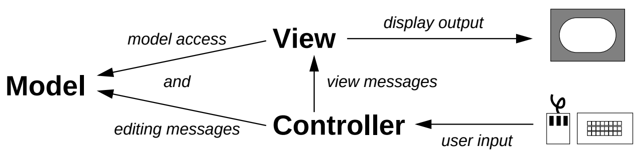
- **Push:** the publisher sends the changed data along with the update message
  Advantages: only one message per subscriber needed.
  Disadvantage: Either the publisher knows for each subscriber what data it needs which increases coupling between publisher and subscriber, or many subscribers receive unnecessary data.
- **Pull**: the subscriber, after receiving the update message, asks the publisher for the specific data he is interested in
  Advantage: Only the necessary amount of data is transferred.
  Disadvantage: a lot of messages have to be exchanged.
- Mixture: the publisher sends hints (“Aspects” in ST terminology) and other parameters along with the update messages

The MVC Pattern

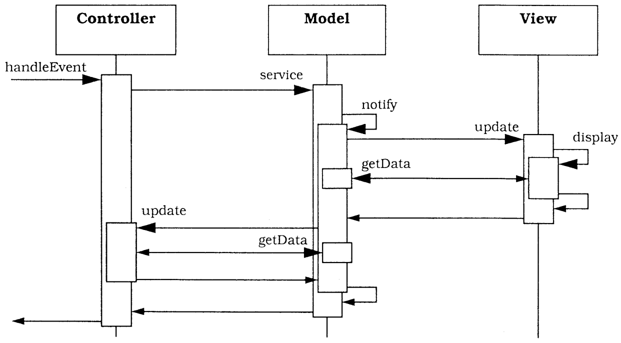
Dependencies:



Other Messages:



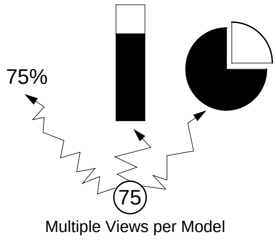
A Standard Interaction Cycle



MVC: Benefits and Liabilities

Benefits:

- Multiple views of the same model
- Synchronized views
- ‘Pluggable’ views and controllers
- Exchangeability of ‘look and feel’



Liabilities:

- Increased complexity
- Potential for excessive number of updates
- Intimate connection between view and controller
- Close coupling of views and controllers to a model
- Inefficiency of data access in view
- Inevitability of change to view and controller when porting

MVC and Smalltalk

- MVC is a pattern and can be also applied with other programming languages.
  - Examples:
    - ET++ User Interface Framework (C++)
    - Swing-Toolkit in the Java Framework

- Nevertheless, the ties between MVC and Smalltalk are exceptionally strong:
  - MVC was invented by a Smalltalker (Trygve Reenskaug)
  - first implemented in Smalltalk-80; the Application Framework of Smalltalk is built around it
  - The first implementations of MVC in Smalltalk have undergone a strong evolution. Newer implementations (for example in VisualWorks) solve many of the problems of the first, straightforward implementations.

Managing Dependents

- Protocol to manage dependents (defined in `Object>>dependents access`):
  - `addDependent:` anObject
  - `removeDependent:` anObject

  ☞ Attention: Storage of Dependents !

- `Object`: keeps all its dependents in a **class** variable `DependentsField` . `DependentsField` is an `IdentityDictionary`, where the keys are the objects themselves and the values are the collections of dependents for the corresponding objects.

- `Model`: defines an **instance** variable `dependents`.
  ☞ access is much more efficient than looking up the dependents in a class variable.

Implementation of Change Propagation

- Change methods are implemented in `Object>>changing`:

```
changed: anAspectSymbol
   "The receiver changed.  The change is denoted by the argument anAspectSymbol.
Usually the argument is a Symbol that is part of the dependent's change protocol,
that is, some aspect of the object's behavior, and aParameter is additional in-
formation.  Inform all of the dependents."

   self myDependents update: anAspectSymbol
```

- Update methods are implemented in `Object>>updating`:

```
update: anAspectSymbol
"Check anAspectSymbol to see if itequals some aspect of interest and if it does,
perform the necessary action"

anAspectSymbol == anAspectOfInterest
   ifTrue: [self doUpdate].
```


Climbing up and down the Default-Ladder

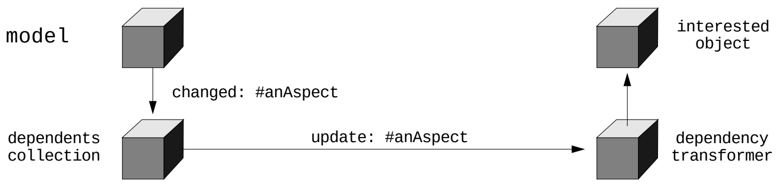
```
changed
   self changed: nil

changed: anAspectSymbol
   self changed: anAspectSymbol with: nil

changed: anAspectSymbol with: aParameter
   self  myDependents  update:  anAspectSymbol  with:  aParameter  from:

               update: anAspectSymbol with: aParameter from: aSender
                  ^self update: anAspectSymbol with: aParameter

               update: anAspectSymbol with: aParameter
                        ^self update: anAspectSymbol

               update: anAspectSymbol
                     ^self
```

Problems ...

- Problems with the Vanilla Change Propagation Mechanism:
  - every dependent is notified about all the changes, even if they are not interested (broadcast).
  - the `update: anAspect` methods are often long lists of tests of `anAspect`. This is not clean object-oriented programming.
  - all the methods changing something have to send `self changed,` since there might just be some dependent that is interested in that change
  - danger of name clashes between apsects that are defined in different models that have to work together (can be solved by using `update:with:from:`)
- General problem: complex objects depending on other complex objects. We need means to be more specific:
  - publisher: send messages only to interested dependents
  - subscriber: being notified directly by a call to the method that handles **that** specific change

Dependency Transformer

- A `DependencyTransformer` is an intermediate object between a model and its dependent. It
  - waits for a specific `update: anAspect` message
  - sends a specific method to a specific object
- A dependent that is only interested in a specific aspect of its model and has a method to handle the update installs a `DependencyTransformer` on its model:




```
model expressInterestIn: anAspect
      for: self
      sendBack: aChangeMessage
```


Inside a Dependency Transformer

- Initializing a `DependencyTransformer`:

```
setReceiver: aReceiver aspect: anAspect selector: aSymbol
   receiver := aReceiver.
   aspect := anAspect.
   selector := aSymbol.
   numArguments := selector numArgs.
   numArguments > 2 ifTrue: [self error: 'selector expects too many arguments']
```

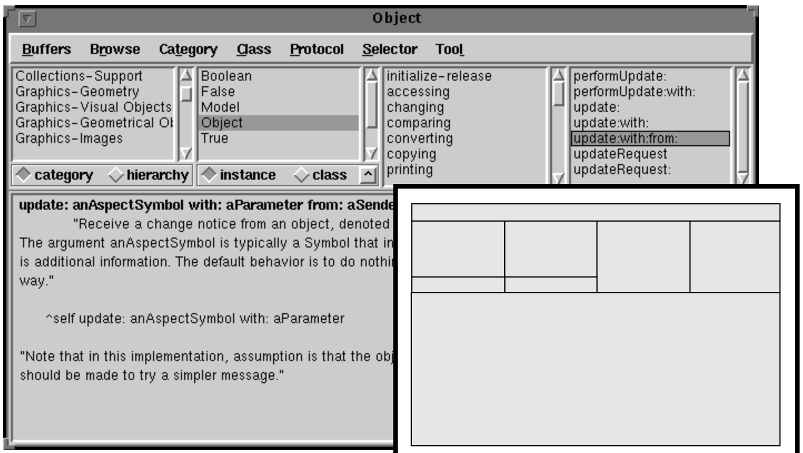
- Transforming an `update:` message:

```
update: anAspect with: parameters from: anObject
   aspect == anAspect ifFalse: [^self].
   numArguments == 0 ifTrue: [^receiver perform: selector].
   numArguments == 1 ifTrue: [^receiver perform: selector with: parameters].
   numArguments == 2 ifTrue: [^receiver perform: selector with: parameters with:
                                                                         anObject]
```

ValueHolder

- A `ValueHolder` is an object that encapsulates a value and allows it to behave like a model, i.e. it notifies the dependents of the model automatically when it is changed.
- Creating a `ValueHolder`:
- Accessing a `ValueHolder`:
- Advantages:
  - change propagation is triggered automatically by the `ValueHolder`; the programmer does not have to do `self changed` any more
  - objects can become dependents only of the values they are interested in (reduces broadcast problem)

A UserInterface Window



The widgets that make up the UI

Widgets

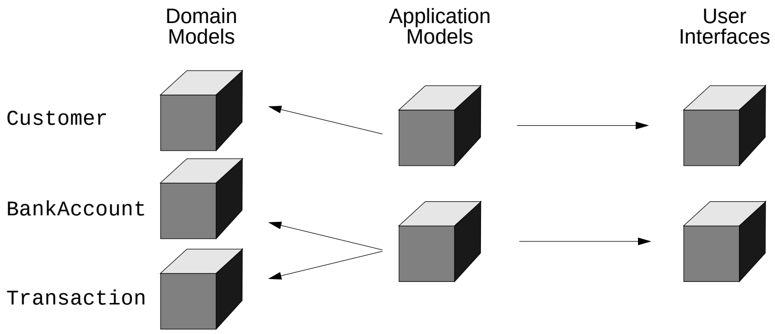
- A widget is responsible for displaying some aspect of a User Interface.
  - A widget can display an aspect of a model
  - A widget can be combined with a controller, in which case the user can modify the aspect of the model displayed by the widget.
- The connection between widgets and the model:
  - Each component of a User Interface is a widget
  - Each component of a model is an attribute or operation
  - Most widgets modify an attribute or start an operation
- The communication between a widget and the model component it represents visually is standardized:

  Value Model Protocol

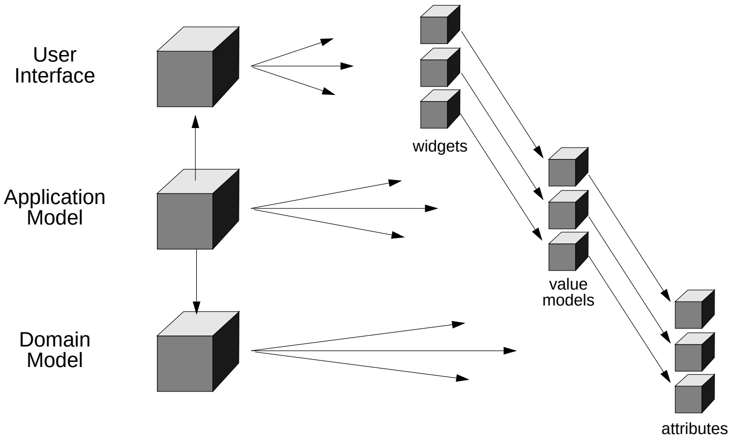
- Each model component is put into an *aspect model*, which can be a `ValueHolder` for example. The Widget deals only with this aspect model.
  - the widget does not have to know any specifics about its model

The Application Model

- An ApplicationModel is a model that is responsible for creating and managing a runtime user interface, usually consisting of a single window. It manages only application information. It leaves the domain information to its aspect models.



The fine-grained Structure of an Application



Bibliography

- E. Gamma et. al.: *Design Patterns*, Addison Wesley, 1995
  - Observer Pattern, pp. 239
- F. Buschmann et. al.: *A System of Patterns. Pattern-Oriented Software Architecture*, Wiley, 1996
  - Model-View-Controller, p. 125
  - Publisher-Subscriber, p. 339
- The VisualWorks Application Framework:
  - VisualWorks Users Guide: *Chapter 18, Application Framework* (available online)
  - Visual Works Cookbook: *Part II, User Interface* (available online)
  - Tim Howard: *The Smalltalk Developer's Guide to VisualWorks*, SIGS Books, 1995

14. Processes and Concurrency

- Concurrency and Parallelism
- Applications of Concurrency
- Limitations
- Atomicity
- Safety and Liveness
- Processes in Smalltalk:
  - Class Process, Process States, Process Scheduling and Priorities
- Synchronization Mechanisms in Smalltalk:
  - Semaphores, Mutual Exclusion Semaphores, SharedQueues
- Delays
- Promises

Concurrency and Parallelism

- A sequential program specifies sequential execution of a list of statements; its execution is called a process. A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes
- A concurrent program can be executed by:
  - *Multiprogramming:* processes share one or more processors
  - *Multiprocessing:* each process runs on its own processor but with shared memory
  - *Distributed processing*: each process runs on its own processor connected by a network to others
- Motivations for concurrent programming:
  - Parallelism for faster execution
  - Improving processor utilization
  - Sequential model inappropriate

Limitations

- Concurrent applications introduce complexity:
  - Safety -> synchronization mechanisms are needed to maintain consistency
  - Liveness -> special techniques may be needed to guarantee progress
  - Non-determinism -> debugging is harder because results may depend on “race conditions”
  - Run-time overhead -> process creation, context switching and synchronization take time

Atomicity

- Programs P1 and P2 execute concurrently:

```
   { x = 0 }
P1: x := x + 1
P2: x := x + 2
   { x = ? }
```

  - What are possible values of x after P1 and P2 complete?
  - What is the *intended* final value of x?

- *Synchronization mechanisms* are needed to restrict the possible interleavings of processes so that sets of actions can be seen as atomic.
- *Mutual exclusion* ensures that statements within a *critical section* are treated atomically.

Safety and Liveness

- There are two principal difficulties in implementing concurrent programs:

Safety - ensuring consistency:

  - *mutual exclusion* - shared resources must be updated atomically
  - *condition synchronization* - operations may need to be delayed if shared resources are not in an appropriate state (e.g, read from an empty buffer)

Liveness - ensuring progress:

  - *No Deadlock* - some process can always access a shared resource
  - *No Starvation* - all processes can eventually access shared resources

- Notations for expressing concurrent computation must address:

1. **Process creation**: how is concurrent execution specified?
2. **Communication**: how do processes communicate?
3. **Synchronization**: how is consistency maintained?

Processes in Smalltalk: Process class

- A Smalltalk system supports multiple independent processes.
- Each instance of class `Process` represents a sequence of actions which can be executed by the virtual machine concurrently with other processes.
- Processes share a common address space (object memory)
- Blocks are used as the basis for creating processes in Smalltalk. The simplest way to create `aProcess` is to send a block the message `#fork`

```
[ Transcript cr; show: 5 factorial printString ] fork
```

- The new process is added to the list of scheduled processes. This process is *runnable* (i.e., scheduled for execution) and will start executing as soon as the current process releases the control of the processor.

Processes in Smalltalk: Process class (II)

- We can create a new instance of class `Process` which is not scheduled by sending the `#newProcess` message to a block:

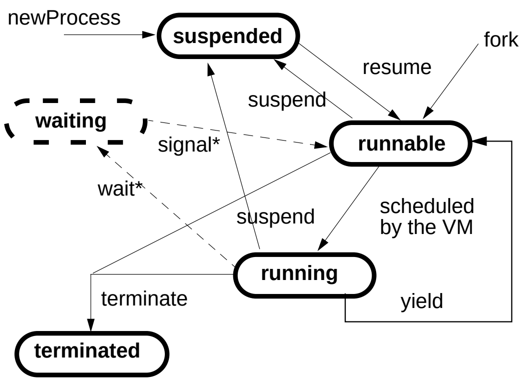
```
| aProcess |
aProcess := [ Transcript cr; show: 5 factorial printString ] newProcess
```

- The actual process is not actually *runnable* until it receives the `#resume` message.
- A process can be created with any number of arguments:

```
aProcess := [ :n | Transcript cr; show: n factorial printString ]
             newProcessWithArguments: #(5).
```

- A process can be temporarily stopped using a `#suspend` message. A suspended process can be restarted later using the `#resume` message.
- A process can be stopped definitely using a message `#terminate`. Once a process has received the `#terminate` message it cannot be restarted any more.

Processes in Smalltalk: Process states



A process may be in one of the five states:

1. suspended
2. waiting
3. runnable
4. running, or
5. terminated

***sent to aSemaphore**

Process Scheduling and Priorities

- Process scheduling is based on priorities associated to processes.
- Processes of high priority run before processes of lower priority.
- Priority values go between 1 and 100.
- Eight priority values have assigned names.

Priority	Name	Purpose
100	timingPriority	Used by Processes that are dependent on real time.
98	highIOPriority	Used by time-critical I/O
90	lowIOPriority	Used by most I/O Processes
70	userInterruptPriority	Used by user Processes desiring immediate service
50	userSchedulingPriority	Used by processes governing normal user interaction
30	userBackgroundPriority	Used by user background processes
10	systemBackgroundPriority	Used by system background processes
1	systemRockBottonPriority	The lowest possible priority

Process Scheduling and Priorities (II)

- Processes are scheduled by the unique instance of class `ProcessorScheduler` called `Processor`.
- A runnable process can be created with an specific priority using the `#forkAt:` message:

```
[ Transcript cr; show: 5 factorial printString ]
      forkAt: Processor userBackgroundPriority.
```

- The priority of a process can be changed by using a `#priority:` message

```
| process1 process2 |
Transcript clear.
process1 := [ Transcript show: 'first'] newProcess.
process1 priority: Processor systemBackgroundPriority.
process2 := [ Transcript show: 'second' ] newProcess.
process2 priority: Processor highIOPriority.
process1 resume.
process2 resume.
```

- The default process priority is `userSchedulingPriority` (50)

The Process Scheduling Algorithm

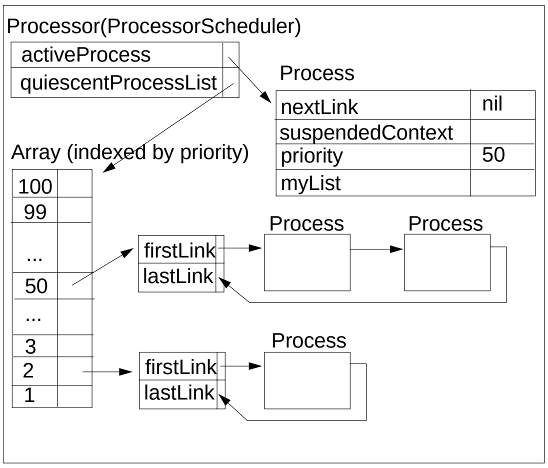
The active process can be identified by the expression:

```
Processor activeProcess
```

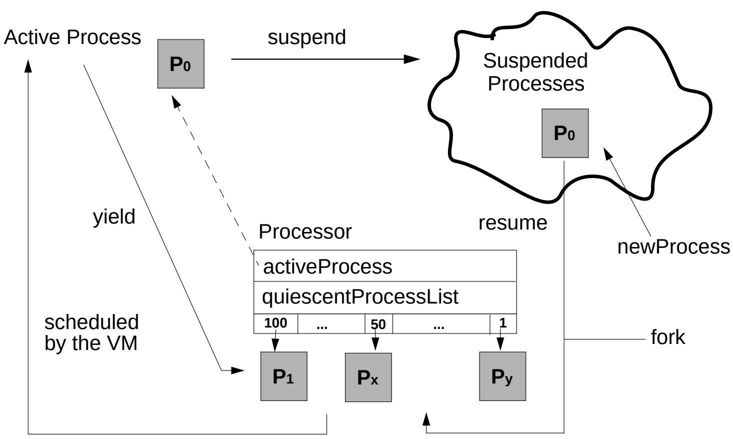
The processor is given to the process having the highest priority.

A process will run until it is suspended, terminated or pre-empted by a higher priority process, before giving up the processor.

When the highest priority is held by multiple processes, the active process can give up the processor by using the message `#yield`.



Process Scheduling



Synchronization Mechanisms

- Concurrent processes typically have references to some shared objects. Such objects may receive messages from these processes in an arbitrary order, which can lead to unpredictable results. Synchronization mechanisms serve mainly to maintain consistency of shared objects.
  - We can calculate the sum of the first N natural numbers:

```
| n |
n := 100000.
[ | i temp |
   Transcript cr; show: ‘P1 running’.
   i := 1. temp := 0.
   [ i <= n ] whileTrue: [ temp := temp + i. i := i + 1 ].
   Transcript cr; show: ‘P1 sum = ‘; show: temp printString ] forkAt: 60.

P1 running
P1 sum is = 5000050000
```


Synchronization Mechanisms (II)

- What happens if at the same time another process modifies the value of n?

```
| n d |
n := 100000.
d := Delay forMilliseconds: 400.
[ | i temp |
   Transcript cr; show: ‘P1 running’.
   i := 1. temp := 0.
   [ i <= n ] whileTrue: [ temp := temp + i.
                           (i = 5000) ifTrue: [ d wait ].
                           i := i + 1 ].
   Transcript cr; show: ‘P1 sum is = ‘; show: temp printString ] forkAt: 60.
[ Transcript cr; show: ‘P2 running’. n := 10 ] forkAt: 50.

P1 running
P2 running
P1 sum is = 12502500
```

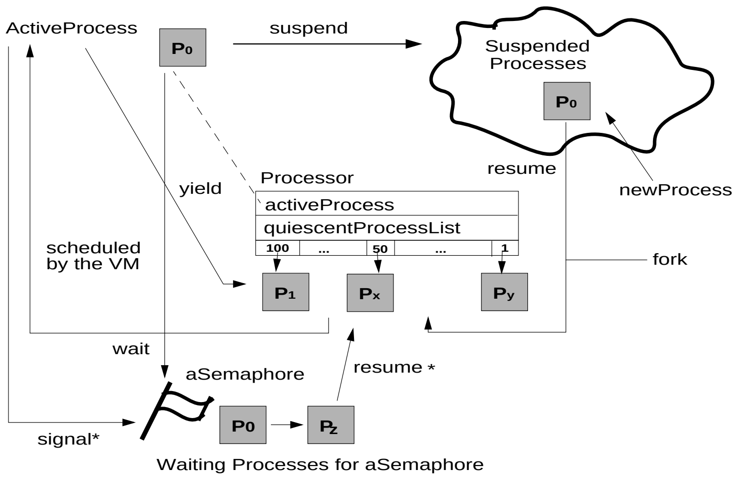

Synchronization using Semaphores

- A semaphore is an object used to synchronize multiple processes. A process waits for an event to occur by sending the message `#wait` to the semaphore. Another process then signals that the event has occurred by sending the message `#signal` to the semaphore.

```
| sem |
Transcript clear.
sem := Semaphore new.
[  Transcript show: ‘The’] fork.
[  Transcript show: ‘quick’. sem wait.
   Transcript show: ‘fox’. sem signal ] fork.
[  Transcript show: ‘brown’. sem signal.
   sem wait. Transcript show: ‘jumps over the lazy dog’; cr ] fork
```

- If a semaphore receives a `#wait` message for which no corresponding `#signal` has been sent, the process sending the `#wait` message is suspended.
- Each semaphore maintains a linked list of suspended processes.
- If a semaphore receives a `#wait` from two or more processes, it resumes only one process for each signal it receives
- A semaphore pays no attention to the priority of a process. Processes are queued in the same order in which they “waited” on the semaphore.

Semaphores



Semaphores for Mutual Exclusion

- Semaphores are frequently used to provide mutual exclusion for a “critical section”. This is supported by the instance method `#critical:`. The block argument is only executed when no other critical blocks sharing the same semaphore are evaluating.

```
| n d sem |
n := 100000.
d := Delay forMilliseconds: 400.
[ | i temp |
   Transcript cr; show: ‘P1 running’.
   i := 1. temp := 0.
   sem critical: [ [ i <= n ] whileTrue: [ temp := temp + i.
                        (i = 5000) ifTrue: [ d wait ].
                        i := i + 1 ]. ].
   Transcript cr; show: ‘P1 sum is = ‘; show: temp printString ] forkAt: 60.
   [  Transcript cr; show: ‘P2 running’. sem critical: [ n := 10 ]] forkAt: 50.
```

- A semaphore for mutual exclusion must start with one extra `#signal`, otherwise the critical section will never be entered. A special instance creation method is provided:

```
Semaphore forMutualExclusion.
```


Synchronization using a SharedQueue

- A `SharedQueue` enables synchronized communication between processes. It works like a normal queue (First in First Out, reads and writes), with the main difference being that `aSharedQueue` protects itself against possible concurrent accesses (multiple writes and/or multiple reads).
- Processes add objects to the shared queue by using the message `#nextPut:` (1) and read objects from the shared queue by sending the message `#next` (3).

```
| aSharedQueue d |
d := Delay forMilliseconds: 400.
aSharedQueue := SharedQueue new.
[ 1 to: 5 do:[:i | aSharedQueue nextPut: i ] ] fork.
[ 6 to: 10 do:[:i | aSharedQueue nextPut: i. d wait ] ] forkAt: 60.
[ 1 to: 5 do:[:i | Transcript cr; show:aSharedQueue next printString] ] forkAt: 60.
```

- If no object is available in the shared queue when the messsage `#next` is received, the process is *suspended.*
- We can query whether the shared queue is empty or not with the message `#isEmpty`

Delays

- Instances of class `Delay` are used to delay the execution of a process.
- An instance of class `Delay` will respond to the message `#wait` by suspending the active process for a certain amount of time.
- The time at which to resume is specified when the delay instance is created. Time can be specified relative to the current time with the messages `#forMilliseconds:` and `#forSeconds:`.

```
| minuteWait |
minuteWait := Delay forSeconds: 60.
minuteWait wait.
```

- The resumption time can also be specified at an absolute time with respect to the system's millisecond clock with the message `#untilMilliseconds:`. Delays created in this way can be sent the message wait at most once.

Promises

- Class `Promise` provides a means to evaluate a block within a concurrent process.
- An instance of Promise can be created by sending the message `#promise` to a block:

```
[ 5 factorial ] promise
```

- The message `#promiseAt:` can be used to specify the priority of the process created.
- The result of the block can be accessed by sending the message value to the promise:

```
| promise |
promise := [ 5 factorial ] promise.
Transcript cr; show: promise value printString.
```

- If the block has not completed evaluation, then the process that attempts to read the value of a promise will wait until the process evaluating the block has completed.
- A promise may be interrogated to discover if the process has completed by sending the message `#hasValue`

15. Classes and Metaclasses - an Analysis

"Some books are to be tasted,
others to be swallowed,
and some few to be chewed and digested"
— Francis Bacon, Of Studies

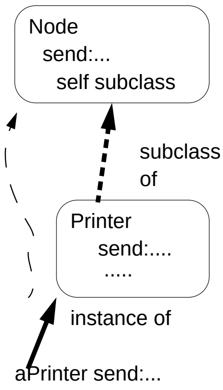
- At first sight, a difficult topic!
- You can live without really understanding them, but metaclasses provide a uniform model, and you will make less errors if you learn how they work, and you will really understand the object model
- Recap on Instantiation
- Recap on Inheritance

The meaning of "Instance of"

- Every object is an instance of a class.
- Every class (except Object) is ultimately a subclass of Object.
- When anObject receives a message, the method is looked up in its class and/or its superclasses.
- A class defines the structure and the behavior of all its instances.
- Each instance possesses its own set of values.
- Each instance shares its behavior with other instances. This behavior is defined in its class, and is accessed via the instance of link.
- Classes are objects: Try to understand

```
OrderedCollection allInstVarNames
OrderedCollection class allInstVarNames
```

- Look at Class class



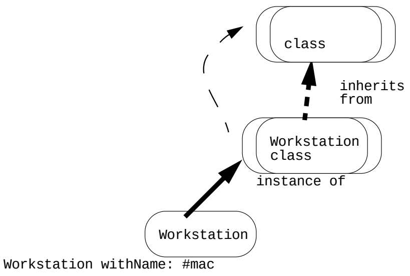
Concept of Metaclass & Responsibilities

- Concept:
  - Everything is an object
  - Every object is instance of exactly one class
  - A class is also an object, and is an instance of its bmetaclass
  - An object is a class if and only if it can create instances of itself.
- Metaclass Responsibilities:
  - instance creation
  - method compilation (different semantics can be introduced)
  - class information (inheritance link, instance variable, ...)
- Examples:

```
Node allSubclasses -> OrderedCollection (WorkStation OutputServer Workstation
   FileServer PrintServer)
PrintServer allInstances -> #()
Node instVarNames -> #('name' 'nextNode')
Workstation withName: #mac -> aWorkstation
Workstation selectors  -> IdentitySet (#accept: #originate:)
Workstation canUnderstand: #nextNode -> true
```

Classes, metaclasses and method lookup

- When anObject receives a message, the method is looked up in its class and/or its superclasses.
- So when aClass receives a message, the method is looked up in its class (a metaclass) and/or its superclass
- Here `Workstation` receives `withName: #mac`
- The method associated with `#withName:` selector is looked up in the class of `Workstation`: `Workstation class`

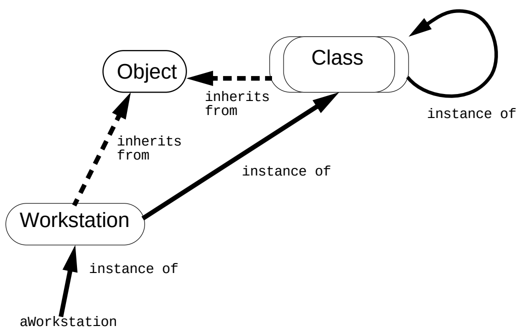


Responsibilities of Object & Class classes

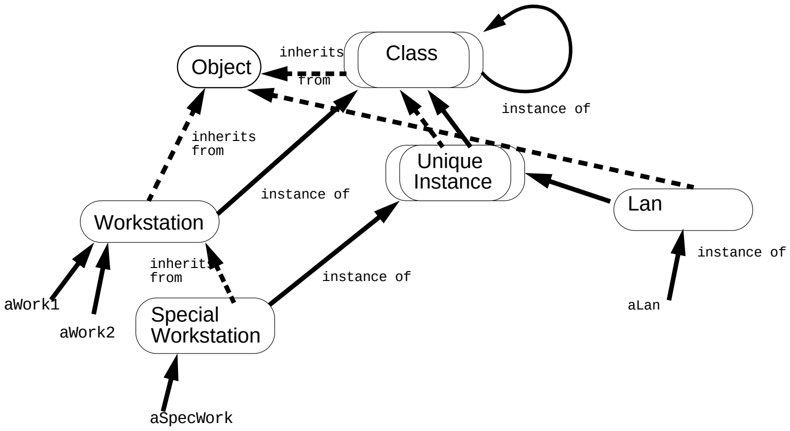
- Object
  - represents the common behavior (like error, halting...) shared by all the instances (final instances and classes)
  - all the classes should inherit ultimately from Object
    - -> Workstation inherits from Node
    - -> Node inherits from Object
- Class
  - represents the common behavior of all the classes (compilation, method storing, instance variable storing)
  - Class inherits from Object because Class is an Object, although a special one -> Class knows how to create instances
  - So all the classes should inherit ultimately from Class

A possible kernel for explicit metaclasses

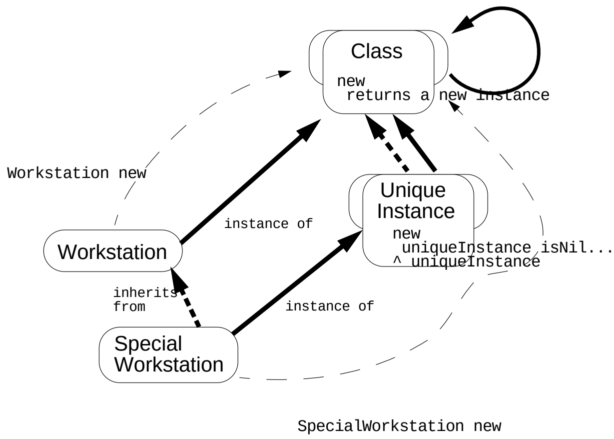
- The kernel of CLOS and ObjVlisp but not the kernel of Smalltalk



Singleton with explicit metaclasses



Deeper into it



Smalltalk Metaclasses in 7 points

- No explicit metaclasses, only implicit non-sharable metaclasses.

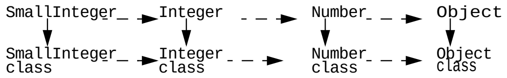
1. Every class is ultimately a subclass of Object (except Object itself)

```
Behavior
    ClassDescription
        Class
        Metaclass
```

2. Every object is an instance of a class = every class is an instance of a class which is its metaclass.

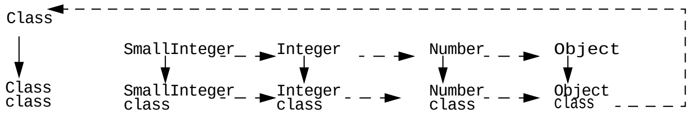
3. Every class is an instance of a metaclass.
   - Every user defined class is the **sole** instance of another class (a metaclass).
   - Metaclasses are system generated so they are unnamed. You can access them by sending the message `#class` to a class.

Smalltalk Metaclasses in 7 points (II)



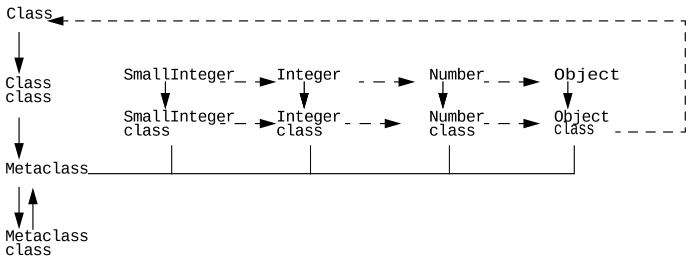
- If X is a subclass of Y then X class is a subclass of Y class.
- But what is the superclass of the metaclass of `Object`?
- The superclass of `Object` class is `Class`

4. All metaclasses are (ultimately) subclasses of `Class`.



- But metaclasses are also objects so they should be instances of a Metaclass

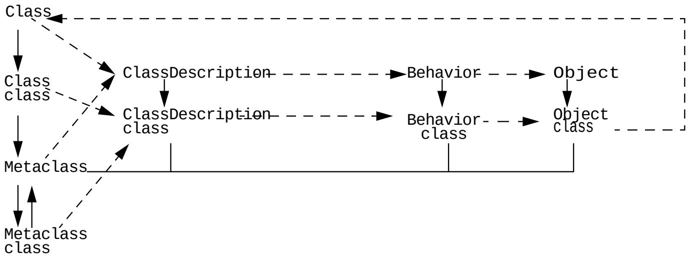
Smalltalk Metaclasses in 7 points (III)



5. Every metaclass is an instance of `Metaclass`. So `Metaclass` is an instance of itself
   - `Object` : common object behavior
   - `Class`: common class behavior (name, multiple instances)
   - `Metaclass`: common metaclass behavior (no name, unique instance)
6. The methods of Class and its superclasses support the behavior common to those objects that are classes.

Smalltalk Metaclasses in 7 points (IV)

7. The methods of instances of `Metaclass` add the behavior specific to particular classes.
   - Methods of instance of Metaclass = methods of “Packet class” = class methods (for example `#withName:`)



- An instance method defined in `Behavior` or `ClassDescription`, is available as a class method. Example: `#new, #new:`

Behavior Responsibilities

- 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 (`#compile:`)
  - instance creation (`#new, #basicNew, #new:, #basicNew:`)
  - class into hierarchy (`#superclass:, #addSubclass:`)
  - accessing (#selectors, #allSelectors, #compiledMethodAt:)
  - accessing instances and variables (`#allInstances, #instVArNames, #allInstVarNames, #classVarNames, #allClassVarNames`)
  - accessing clas hierarchy (`#superclass, #allSuperclasses, #subclasses, #allSubclasses`)
  - testing (`#hasMethods, #includesSelector, #canUnderstand:, #inheritsFrom:, #isVariable`)

ClassDescription Responsibilities

- `ClassDescription` adds a number of facilities to basic `Behavior`:
  - named instance variables
  - category organization for methods
  - the notion of a name of this class (implemented as subclass responsibility)
  - the maintenance of the Changes set, and logging changes on a file
  - 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`.
- Subclasses must implement `#addInstVarName:` and `#removeInstVarName:`
- Instance Variables:
  - instanceVariables<Array of: String> names of instance fields
  - organization <ClassOrganizer> provides organization of message protocol

Metaclass and Class Responsibilities

- Metaclass
  - initialization of class variables
  - creating initialized instances of the metaclass's sole instance
  - instance creation (`#subclassOf:`)
  - metaclass instance protocol (`#name:inEnvironment:subclassOf:`....)
- Class
  - `Class` adds naming for class
  - `Class` adds the representation for classVariable names and shared pool variables (`#addClassVaraNames, #addSharedPool:, #initialize`)

16. Common Mistakes and Debugging

- Preventing: Most Common Mistakes
- Curing: Debugging Fast (from ST Report July 93)
- Extras

Common Beginner Bugs

- `true` is the boolean value, `True` its class. Which one is correct?

```
Book>>initialize
    inLibrary := True

Book>>initialize
    inLibrary := true
```

- `nil` is not an acceptable receiver for `ifTrue:`
- `whileTrue` receiver must be a block

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

- Before creating a class, check if it already exists. This is (sigh) a weakness of the system

```
Object subclass: #View
```


Common Beginner Bugs (II)

- In a method `self` is returned by default. Do not forget **^** for returning something else.

```
Packet>>isAddressedTo: aNode
    ^ self addressee = aNode name
```

- In a `#new` method do not forget the ^ to return the newly created instance

```
Packet class>>new
    super new initialize
```

- The above code returns the class `Packet` and not the newly created instance. The correct code is

```
Packet class>>new
    ^ super new initialize
```

Common Beginner Bugs (III)

- In a `new` method do not forget to use `super` or to invoke `basicNew` to create the new instance.
  - The following Example loops:

```
Packet class>> new
  ^self new initialize
```

  - The correct code is:

```
Packet class>> new
  ^ self basicNew initialize
  "or ^ super new initialize"
```

- Before redefining new as follows:

```
Packet class>>new
  ^super new initialize
```

  - check if this is not already done by super. If so, `initialize` will be called twice!

Instance Variable Access in Class Method

- Do not try to access instance variables to initialize them in the `new` method. You do not have the right. The `new` method can only access class instance variables and classVariables.
  - -> Define and invoke an `initialize` method on instances.
  - Example: Do not write

```
Packet class>>send: aString to: anAddress
  contents := aString.
  addressee := anAddress
```

  - Instead create an instance and invoke instance methods

```
Packet class>>send: aString to: anAddress
  self new contents: aString; addressee: anAddress
```


Common Beginner Bugs - Assignment

- Do not try to assign a value to a method argument

```
setName: aString
  aString := aString, 'Device'.
  name := aString
```

- Do not assign to a class, it will damage your system

```
OrderedCollection := 2
```

- Do not try to modify `self` and `super`

Common Beginner Bugs - Redefinition

- **Never** redefine **basic**-methods (`#==, #basicNew, #basicNew:, #basicAt:, #basicAt:Put:`...)
- **Never** redefine `#class`
- Redefine `#hash` when you redefine `#=` so that if a = b then a hash = b hash

```
Book>>=aBook
   ^self title = aBook title & (self author = aBook author)

Book>>hash
   ^self title hash bitXor: self author hash
```


Common Beginner Bugs - Collections

- `#add:` returns the argument and not the receiver, so use `yourself` to get the collection back.
- Do not forget to specialize `#copyEmpty` when adding named instance variables to a subclass which has **indexed** instance variables (subclasses of Collection)
- Never iterate over a collection which the iteration somehow modifies.

```
timers do:[:aTimer| aTimer isActive ifFalse: [ timers remove: aTimer]]
```

  - **First Copy** the collection

```
timers copy do:[:aTimer| aTimer isActive ifFalse: [ timers remove: aTimer]
```

- Take care, since the iteration can involve various methods and modifications which may not be obvious!

Use of Accessors: Protect your Cients

- The literature says: “Access instance variables using methods”

```
Schedule>>initialize
   tasks := OrderedCollection new.
Schedule>>tasks
   ^tasks
```

- However, accessors methods should be PRIVATE by default.
- If accessors would be public, a client could write

```
ScheduleView>>addTaskButton
   ...
   model tasks add: newTask
```

  - What happens if we change the representation of tasks? If `tasks` is now a dictionary everything will break.
  - Provide an adding method

```
Schedule>>addTask: aTask
   tasks add: aTask
ScheduleView>>addTaskButton
   ...
   model addTask: newTask
```

Debugging - Hints

- Basic Printing

```
Transcript cr; show: 'The total= ', self total printString.
```

  - Use a global or a class to control printing information

```
Debug ifTrue:[Transcript cr; show: 'The total= ', self total printString]
Debug > 4 ifTrue:[Transcript cr; show: 'The total= ', self total printString]
Debug print:[Transcript cr; show: 'The total= ', self total printString]
Smalltalk removeKey: #Debug
```

- Inspecting

```
Object>>inspect
```

  - You can create your own inspect method

```
MyInspector new inspect: anObject
```

- Naming: useful to add an id for debugging purposes

Debugging - Where am I?

- Identifying the current context

```
"if this is not a block"
Transcript show: thisContext printString; cr.
Debug ifTrue:[ "use this expression in a block"
   Transcript show: thisContext sender home printString; cr]
```

- Audible Feedback

```
Screen default ringBell
```

- Catching It in the Act

```
<Ctrl-C> (VW2.5) <Ctrl-Shift-C> Emergency stop
<Ctrl-Y> (VW3.0) <Ctrl-Shift-C> Emergency stop
```

- Suppose that you cannot open a debugger

```
Transcript cr; show: (Notifierview shortStackFor: thisContext ofSize: 5)
```

  - Or in a file

```
|file|
file := 'errors' asFilename appendStream.
file cr; nextPutAll: (NotifierView shortStackFor: thisContext ofSize: 5).
file close
```


Debugging - Source Inspection

- Source Code for Blocks

```
aBlockClosure method getSource
aMethodContext sourceCode
```

- Decompiling a Method
  - Shift + select the method in the browser
  - Interesting for modifying literals or fixing MethodWrapper bugs:

```
initialize
arrayConst := #(1 2 3 4)
```

  - then somebody somewhere does

```
arrayConst at:1 put:100
```

  - So your array is polluted. Note that if you recompile the method the original contents of the literal array are restored. So always consider returning copies of your literals.

- Entry Points
  - How is a window opened or what happens when the menu is invoked? Look into `LauncherView` and `UIVisualILauncher` implementors of '*enu*'

Debugging - Where am I going?

- Breakpoints

```
self halt.
self error: ' invalid'
```

- Conditional halt

```
i > 10 ifTrue:[self halt]
InputState default shiftDown ifTrue:[self halt]
InputState default altDown ifTrue:[self halt]
InputState default metaDown ifTrue:[self halt]
```

- In a controller:

```
self sensor shiftDown ifTrue:[self halt]
```

- Slowing Down Actions: useful for complex graphics

```
Cursor wait showWhile: [(Delay forMilliseconfs: 800) wait]
```

  - (Do not forget the wait) until a mouse button is clicked.

```
Cursor crossHair showWhile:
    [ScheduledControllers activeController sensor waitNoButton; waitClickButton]
```


Debugging - How do I get out?

```
1 <CTRl+Shift-C or Y> Emergency Debugger
2 ObjectMemory quit
3 <ESC> to evaluate the expression
```

- An Advanced Emergency Procedure: recompile the wrong method if you know it!

```
aClass compile: 'methodname methodcode' classified: 'what you want'
example:
Controller compile: 'controlInitialize ^self' classified: 'basic'
```

- Graphical Feedback
  - Where the cursor is:

```
ScheduledControllers activeController sensor cursorPoint
```

  - Position the cursor explicitly

```
ScheduledControllers activeController sensor cursorPoint: aPoint
Rectangle fromUser
```

  - Indicating an area with a filled rectangle

```
ScheduledControllers activeController view graphicsContext
    display Rectangle: (0@0 extent: 10@100)
```


Debugging - Files in VW

```
ExternalStream classPool at: #openStreams
```

- How do you ensure that an open file will be closed in case of an error?
  - Use `#valueNowOrOnUnwindDo:` or `#valueOnUnwindDo:`

```
|stream|
[ stream := (Filename named: aString) readStream.
    ...
] valueNowOrOnUnwindDo: [stream close].

BlockClosure>>valueOnUnwindDo: aBlock
"Answer the result of evaluating the receiver. If an exception would cause
the evaluation to be abandoned, evaluate aBlock. "

BlockClosure>>valueNowOrOnUnwindDo: aBlock
"Answer the result of evaluating the receiver. If an exception would cause the
evaluation to be abandoned, evaluate aBlock.  The logic for this is in Exception.
If no exception occurs, also evaluate aBlock."
```

17. The Internal Structure of Objects

- Smalltalk gives to the programmer the illusion of uniformity
  - for example SmallIntegers are defined as any other object but in memory they are different than objects. In that case the object pointer represents the SmallInteger
- In the memory representation Smalltalk objects can be of
  - pointer type
  - non-pointer type
  - index type (e.g., #(1 2 3) at: 2)
  - non-index type (e.g., aPacket name)
  - immediate type
- This difference is transparent for the programmer's daily job, but if we want to do some optimizations, performance and memory analysis.... how can we compute the size in bytes of an object?

Three Ways to Create Classes

- Non indexable, pointer

```
Object subclass: #Packet
    instanceVariableNames: 'contents addressee originator '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Demo-LAN'
```

- Indexable pointer

```
ArrayedCollection variableSubclass: #Array
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Collections-Arrayed'
```

- Indexable, non pointer

```
LimitedPrecisionReal variableByteSubclass: #Float
    instanceVariableNames: ''
    classVariableNames: 'Pi RadiansPerDegree '
    poolDictionaries: ''
    category: 'Magnitude-Numbers'
```

- It is not possible to define named instance variables

Let there be Code

- Identifying subclass:

```
| collection |
collection := SortedCollection new.
Smalltalk allBehaviorsDo: [:each ||boolean|
    boolean := each isMeta not and: [each isObsolete not].
    boolean := boolean and: [each isFixed].
    boolean ifTrue: [collection add: each name]].
^collection
```

- Identifying variableSubclass:

```
boolean := each isMeta not and: [each isObsolete not].
boolean := boolean and: [each isPointers].
boolean := boolean and: [each isVariable].
boolean ifTrue: [collection add: each name]]
```

- Identifying variableByteSubclass:

```
boolean := each isMeta not and: [each isObsolete not].
boolean := boolean and: [each isBits].
boolean := boolean and: [each isVariable].
boolean ifTrue: [collection add: each name]]
```

Format and other

- The information for distinguishing between these three type is stored in the format instance variable of Behavior.

```
Behavior>>isBits
"Answer whether the receiver contains just bits (not pointers)."
^format noMask: self pointersMask

Behavior>>hasImmediateInstances      immediate type object?
Behavior>>isFixed                    non-indexable type object?
Behavior>>isPointers                 pointers type object?
Behavior>>isVariable                 indexable type object?
```

pointer type [isPointers]
indexable type [isVariable] variableSubclass:
non-index type [isFixed] subclass:
non-pointer [isBits]
index type [isVariable] variableByteSubclass:
non-index type [isFixed] subclass:
immediate [hasImmediateInstances] subclass:

Object size in bytes

```
objectSizeInBytes: anObject
    |bytesInOTE bytesInOOP aClass indexableFieldSize instVarFieldSize size|
    bytesInOTE := ObjectMemory current bytesPerOTE.
    bytesInOOP := ObjectMemory current bytesPerOOP.
    aClass := anObject class.
    aClass isPointers
        ifTrue:
            [instVarFieldSize := aClass instSize * bytesInOOP.
            aClass isVariable
                ifTrue: [indexableFieldSize := anObject basicSize * bytesInOOP]
                ifFalse: [indexableFieldSize := 0]]
        ifFalse:
            [instVarFieldSize := 0.
            aClass isVariable
                ifTrue: [indexableFieldSize := anObject basicSize +
                                    (bytesInOOP -1) bitAnd: bytesInOOP negated]
                ifFalse:[indexableFieldSize := 0]].
    size := bytesInOTE + instVarFieldSize + indexableFieldSize.
    ^size
```


Analysis

- OTE (ObjectTable Entry) = 12 bytes: OTE is a description of an Object (class, iv, hash, gc flags, ...)
- OOP (Object Oriented Pointer) = 4 bytes
- Pointers Type

```
Internals new objectSizeInBytes: WorkStation new
    pointer, instSize = 3 (dependents name nextNode) * 4 = 12
    not indexable
Internals new objectSizeInBytes: (WorkStation new name: #abc)
    idem, because not recursive
Internals new objectSizeInBytes: 1@2
    12 + 2 * 4 = 20 bytes
```

- Indexable and Pointers Type

```
Internals new objectSizeInBytes: (OrderedCollection new: 10)
    OrderedCollection new: 10
        = 2 inst variable and 10 indexes
    class instSize = 2 * 4
    basicSize = 10 * 4
    = 60 bytes
```

Analysis (II)

- Indexable pure

```
Internals new objectSizeInBytes: Float pi
   4 indexed variable * 4 = 16 bytes
```

- Non pointer, non Index = immediate, but an immediate type object has no object table entry. The immediate object is stored into the OOP.

```
Internals new objectSizeInBytes: 1
   = 12 bytes, but the code should use isImmediate
```

18. Blocks and Optimization

- Recall:

```
[ :x :y | |tmp| ...]
   value
   value:
   value: value:
   value: value: value:
   valueWithArguments:
```

- In VisualWorks there are four types of blocks:
  - Full Blocks
  - Copying Blocks
  - Clean Blocks
  - Inlined Blocks

- The programmer does not have to explicitly mention which one is needed. This is inferred by the compiler. However, knowing the subtle differences allows the programmer to write more efficient code.

Full Blocks

- Read and assign temporary variables.
- Block containing explicit return ^.
- Compiled in a BlockClosure.
- Evaluation by the creation of an **explicit** MethodContext or BlockContext object instead of using a pseudo-object contained in the stack.
- Most costly

Instead of:

```
m1: arg1
   arg1 isNil
      ifTrue: [^ 1]
      ifFalse: [^ 2]
```

Better:

```
m1: arg1
   ^ arg1 isNil
         ifTrue: [1]
         ifFalse: [2]
```


Copying Blocks

- Read temporary variables but do not assign them.
- No explicit return.
- Access instance variables of self and assign them.
- Not compiled into a BlockClosure.
- They are compiled by copying every access into the block, thus avoiding explicit references to a context where the copied variables appear.
- Their arguments and temporaries are merged into the enclosing method's context as "compiler-generated temporaries".

Clean Blocks

- Contain only reference block temporary variables or global variables.
- No reference to self or to instance variables.

```
nodes do: [:each | each name = #stef]
nodes select: [:each | each isLocal]
```


Inlined Blocks

- Code of certain methods, like whileFalse: ifTrue:, is directly inlined into the code of the calling method.
- The literal blocks (without arguments) passed as argument to such methods are also inlined in the byte-code of the calling method.
- Inlined methods are whileTrue, whileTrue:, whileFalse, whileFalse:, and: or:, ifTrue:, ifFalse:, ifTrue:ifFalse:, ifFalse:ifTrue:, to:do:, to:do:by:
- Look in MessageNode>>transform* methods to see the inlining

testInLined
```
1 to: 5 do: [:x| ]
```

Compiled into :
```
| t1 |
t1 := 1.
[t1 <= 5] whileTrue: [t1 := t1 + 1].
```

- But no BlockClosure is created (look into the byte codes)

Full to Copy

- Instead of:
```
|t|
[:x | t := x foo] value: 1.
t := t * 2.
^t
```
  - The reference to t inside the block makes it at least a copying block.
  - t := makes it full.

- With the following we have a clean block.
```
|t|
t := [:x | x foo] value:1.
t := t * 2.
^t
```

Contexts

- Full blocks are evaluated in a separate context.
- The following code evaluates to false:

```
|outerContext answer|
outerContext := thisContext.
(1 to: 1) do: [:i | answer := thisContext == outerContext].
^answer
```

- But the following evaluates to true because: to:do: is an inlined block

```
|outerContext answer|
outerContext := thisContext.
1 to: 1 do: [:i | answer := thisContext == outerContext].
^answer
```

- So it is better to use to:do: than (to:) do:

inject:into:

- Instead of:

```
|maxNumber|
maxNumber := 0.
#(1 2 43 56 2 49 3 2 0 ) do: [:each| maxNumber := each max: maxNumber].
^maxNumber
```

- Write

```
#(1 2 43 56 2 49 3 2 0 ) inject: 0 into: [:maxNumber :ele| maxNumber max: ele]
```

- no need for a temporary variable
- full block becomes a clean block

About String Concatenation

- str1 , str2 creates a new structure in which str1 and str2 elements are stored

```
SequenceableCollection>>, aSequenceableCollection
   "Answer a copy of the receiver concatenated with the argument,
   a SequenceableCollection."

   ^self copyReplaceFrom: self size + 1
        to: self size
        with: aSequenceableCollection
SequenceableCollection>>copyReplaceFrom: start to: stop with: replacementCollection
   "Answer a copy of the receiver satisfying the following conditions:
   .."
```

Streams, Blocks and Optimization

(from Alan Knight)

- Suppose that we want to concatenate a pretty long list of strings, for example the keys of the Smalltalk dictionary.

```
|bigString|
bigString := String new.
Smalltalk keys do: [:aString | bigString := bigString, aString].
```

- Here the assignment of bigString leads to a Full Block
- We can suppress the assignment like that and thus obtain a clean block

```
|aStream|
aStream:= WriteStream on: String new.
Smalltalk keys do: [:aString | aStream nextPutAll: aString].
```

- inject:into: allows us to suppress the reference to variables that are outside the block and to obtain a clean block.

```
|aStream|
aStream:= WriteStream on: String new.
Smalltalk keys inject: aStream into: [:cumul :aString| cumul nextPutAll: aString. cumul].
```


Streams, Blocks and Optimization (II)

- Now if we use a stream for the Smalltalk keys we can avoid an iteration method. With whileFalse: that is inlined the block itself will be inlined.

```
|aReadStream aWriteStream|
aReadStream := ReadStream on: Smalltalk keys asArray.
aWriteStream := WriteStream on: String new.
[aReadStream atEnd] whileFalse: [aWriteStream nextPutAll: a ReadStream next].
```

Optimization Yes, but Readibility First

BlockClosure Class Comments

- Instance Variables:

```
method <CompiledBlock>
outerContext <Context | nil>
copiedValues <Object | Array | nil>
```

- There are currently three kinds of closures:

1. "Clean" closure with no references to anything from outer scopes. A clean closure has outerContext = nil and copiedValues = empty Array.
2. "Copying" closure that copies immutable values from outer scopes when the closure is created. A copying closure has outerContext = nil and copiedValues = Object or Array.
3. "Full" closure that retains a reference to the next outer scope. A full closure has outerContext ~= nil and copiedValues = nil.

- As an optimization, copiedValues holds the single copied value if there is exactly one, or an Array of values if there is more than one. Note that if there is a single copied value, the value being copied can be nil, so testing for nil in copiedValues is not a reliable means of classifying closures. The way to check whether a closure has copied values is to ask its method whether numCopiedValues > 0.

19. Advanced Blocks

- VM represents the state of execution as Context objects
  - for method MethodContext
  - for block BlockContext

- aContext contains a reference to
  - the context from which it is invoked,
  - the receiver
  - arguments
  - temporaries in the Context

- We call home context the context in which a block is defined

Lexical Scope

- Arguments, temporaries, instance variables are lexically scoped in Smalltalk
- These variables are bound in the context in which the block is defined and not in the context in which the block is evaluated

```
Test>>testScope
   "self new testScope"
   |t|
   t := 15.
   self testBlock: [Transcript show: t printString]

Test>>testBlock:aBlock
   |t|
   t := 50.
   aBlock value

Test new testBlock
   -> 15 and not 50
```


Returning from a Block

- ^ should be the last statement of a block body

```
   [ Transcript show: 'two'.
   ^ self.
   Transcript show: 'not printed']
```

- ^ return exits the method containing it.

```
test
   "self new test"
   Transcript show: 'one'.
   1 isZero
      ifFalse: [ 0 isZero ifTrue: [ Transcript show: 'two'.
                              ^ self]].
   Transcript show: ' not printed'

-> one two
```

Returning From a Block (II)

- Taking returning as a differenciator
  - Simple block [:x :y| x *x. x + y] returns the value of the last statement to the method that send it the message value
  - Continuation blocks [:x :y| ^ x + y] returns the value to the method that activated @@not clear activated@@ its homeContext
- As a block is always evaluated in its homeContext, it is possible to attempt to return from a method which has already returned using other return. This runtime error condition is trapped by the VM.

```
Object>>returnBlock
   ^[^self]
Object new returnBlock
```

-> Exception

```
|b|
b:= [:x| Transcript show: x. x].
b value: ‘ a’. b value: ‘ b’.
b:= [:x| Transcript show: x. ^x].
b value: ‘ a’. b value: ‘ b’.
```

- Continuation blocks cannot be executed several times!

Example of Block Evaluation

```
Test>>testScope
   "self new testScope"
   |t|
   t := 15.
   self testBlock: [Transcript show: t printString.
                   ^self ]

Test>>testBlock:aBlock
   |t|
   t := 50.
   aBlock value.
   self halt.

Test new testBlock
   -> 15 and not halt!!
```


Creating an Escape Mechanism

```
|val|
val := [:exit |
         |goSoon|
         goSoon := Dialog confirm: 'Exit now?'.
         goSoon ifTrue: [exit value: 'Bye'].
         Transcript show: 'Not exiting'.
         'last value'] valueWithExit.
Transcript show: val
```

- yes -> print ‘Bye’
- no -> print ‘Not Exiting’ last value

```
BlockClosure>>valueWithExit
   ^self value: [:arg| ^arg ]
```

Part III - Design Considerations

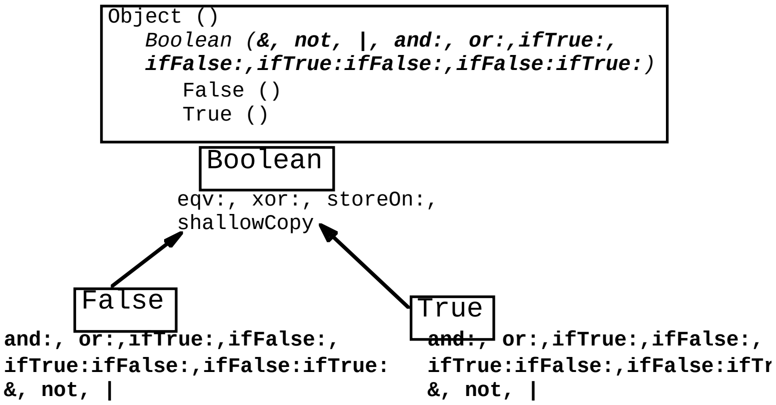
- Abstract Classes
- Elements of Design
- Elementary Design Issues
- Selected Idioms
- Selected Design Patterns

20. Abstract Classes

- Should not be instantiated (abstract in Java).
- Defines a protocol common to a hierarchy of classes that is independent from the representation choices.
- A class is considered as abstract as soon as one of the methods to which it should respond to is not implemented (can be a inherited one).
- Deferred methods send the message `self subclassResponsibility`.
- Depending of the situation, override `#new` to produce an error.
- Abstract classes are not syntactically different from instantiable classes, BUT a common convention is to use class comments: So look at the class comment and write in the comment which methods are abstract and should be specialized.
  - Advanced tools check this situation and exploit it.

```
"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:"
```


Case Study - Boolean, True and False



Case Study - Boolean, True and False (II)

- Abstract method

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

- Concrete method defined in terms of an abstract method

```
Boolean>>xor: aBoolean
    "Exclusive OR.  Answer true if the receiver is not equivalent to aBoolean."
    ^(self == aBoolean) not
```

  - When `#not` will be defined, `#xor:` is automatically defined

- Note that 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.

Case Study - Boolean, True and False (III)

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

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

False>>ifTrue: trueBlock ifFalse: falseBlock
    "Answer the value of falseBlock. This method is typically not invoked because
    ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."
    ^falseBlock value

True>>ifTrue: trueBlock ifFalse: falseBlock
    "Answer the value of trueBlock. This method is typically not invoked because
    ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."
    ^trueAlternativeBlock value
```


Case Study - Magnitude

```
1 > 2  =  2 < 1 = false
```

```
Magnitude>> < aMagnitude
    ^self subclassResponsibility
Magnitude>> = aMagnitude
    ^self subclassResponsibility
Magnitude>> <= aMagnitude
    ^(self > aMagnitude) not
Magnitude>> > aMagnitude
    ^aMagnitude < self
Magnitude>> >= aMagnitude
    ^(self < aMagnitude) not
Magnitude>> between: min and: max
    ^self >= min and: [self <= max]
```

```
1 <= 2    = (1 > 2) not
          = false not
          = true
```


Case Study - Date

```
Date>>< aDate
    "Answer whether the argument, aDate, precedes the date of the receiver."

    year = aDate year
        ifTrue: [^day < aDate day]
        ifFalse: [^year < aDate year]

Date>>= aDate
     "Answer whether the argument, aDate, is the same day as the receiver. "

    self species = aDate species
        ifTrue: [^day = aDate day & (year = aDate year)]
        ifFalse: [^false]

Date>>hash
    ^(year hash bitShift: 3) bitXor: day
```

21. Elements of Design

- Class definition
- Instance initialization
- Enforcing the instance creation
- Instance / Class methods
- Instance variables / Class instance variables
- Class initialization
- Law of Demeter
- Factoring Constants
- Abstract Classes
- Template Methods
- Delegation
- Bad Coding Style

A First Implementation of Packet

```
Object subclass: #Packet
    instanceVariableNames: ‘contents addressee originator ‘
    classVariableNames: ‘’
    poolDictionaries: ‘’
    category: ‘Lan-Simulation’
```

- One instance method

```
Packet>>printOn: aStream
    super printOn: aStream.
    aStream nextPutAll: ‘ addressed to: ‘; nextPutAll: self addressee.
    aStream nextPutAll: ‘ with contents: ’; nextPutAll: self contents
```

- Some Accessors

```
Packet>>addressee
    ^addressee

Packet>>addressee: aSymbol
    addressee := aSymbol
```


Packet CLASS Definition

- Packet Class is **Automatically** defined

```
Packet class
    instanceVariableNames: ''
```

- Example of instance creation

```
Packet new addressee: # mac ; contents: ‘hello mac’
```

Fragile Instance Creation

```
Packet new addressee: # mac ; contents: 'hello mac'
```

- If we do not specify a contents, it breaks!

```
|p|
p := Packet new addressee: #mac.
p printOn: aStream -> error
```

- Problems of this approach:
  - responsibility of the instance creation relies on the clients
  - can create packet without contents, without address
  - instance variable not initialized -> error (for example, `printOn:`) -> **system fragile**
- Solutions:
  - Automatic initialization of instance variables
  - Proposing a solid interface for the creation
  - Lazy initialization

Assuring Instance Variable Initialization

- **Problem:** By default `#new` class method returns instance with uninitialized instance variables. Moreover, `#initialize` method is not automatically called by creation methods `#new/new:`
  - How to initialize a newly created instance ?
- **Solution:** Define an instance method that initializes the instance variables and override `#new` to invoke it.

```
1       Packet class>>new                          Class Method
2          ^ super new initialize

3       Packet>>initialize                         Instance Method
           super initialize.
4          contents := 'default message'
```

```
Packet new (1-2) -> aPacket initialize (3-4) -> returning anInitializedPacket
```

- **Reminder:** You cannot access instance variables from a class method like `#new`

Strengthen Instance Creation Interface

- **Problem:** A client can still create `aPacket` without address.
- **Solution:** Force the client to use the class interface creation.
  - Providing an interface for creation and avoiding the use of `#new`

    ```
    Packet send: 'Hello mac' to: #Mac
    ```
  - First try:

```
Packet class>>send: aString to: anAddress
   ^ self new contents: aString ; addressee: anAddress
```

Other Instance Initialization

step 1. SortedCollection sortBlock: [:a :b| a name < b name]

```
SortedCollection class>>sortBlock: aBlock
  "Answer a new instance of SortedCollection such that its elements are sorted
   according to the criterion specified in aBlock."
   ^self new sortBlock: aBlock                                  Class method
```

step 2. self new = aSortedCollection

step 3. aSortedCollection sortBlock: aBlock   Instance method

step 4. returning the instance aSortedCollection

step 1. OrderedCollection with: 1

```
Collection class>>with: anObject
   "Answer a new instance of a Collection containing anObject."
   | newCollection |
   newCollection := self new.
   newCollection add: anObject.
   ^newCollection
```


Lazy Initialization

- When some instance variables are:
  - not used all the time
  - consuming space, difficult to initialize because depending on other
  - need a lot of computation
  - Use lazy initialization based on accessors
  - Lazy initialization should be used consistently!

- A lazy initialization scheme with default value

```
Packet>>contents
   contents isNil
      ifTrue: [contents := ‘no contents’]
   ^ contents
```

- A lazy initialization scheme with computed value

```
Dummy>>ratioBetweenThermonuclearAndSolar
   ratio isNil
      ifTrue: [ratio := self heavyComputation]
   ^ ratio
```


Providing a Default Value

- The case of SortedCollection

```
OrderedCollection variableSubclass: #SortedCollection
   instanceVariableNames: 'sortBlock '
   classVariableNames: 'DefaultSortBlock '

SortedCollection class>>initialize
   DefaultSortBlock := [:x :y | x <= y]

SortedCollection>>initialize
   "Set the initial value of the receiver's sorting algorithm to a default."
   sortBlock := DefaultSortBlock

SortedCollection class>>new: anInteger
   "Answer a new instance of SortedCollection. The default sorting is a <= comparison on elements."
   ^(super new: anInteger) initialize

SortedCollection class>>sortBlock: aBlock
   "Answer a new instance of SortedCollection such that its elements
   are sorted according to the criterion specified in aBlock."
   ^self new sortBlock: aBlock
```

Invoking per default the creation interface

```
OrderedCollection class>>new
  "Answer a new empty instance of OrderedCollection."

  ^self new: 5
```


Forbidding new

- **Problem:** We can still use `#new` to create fragile instances
- **Solution:** `#new` should raise an error!

```
Packet class>>new
   self error: 'Packet should only be created using send:to:'
```

- But we still have to be able to create instance!

```
Packet class>>send: aString to: anAddress
   ^ self new contents: aString ; addressee: anAddress
```

-> raises an error

```
Packet class>>send: aString to: anAddress
   ^ super new contents: aString ; addressee: anAddress
```

-> bad style: link between class and superclass dangerous in case of evolution

- Solution: use `basicNew` and `basicNew:`

```
Packet class>>send: aString to: anAddress
   ^ self basicNew contents: aString ; addressee: anAddress
```


Class Methods - Class Instance Variables

- Classes (`Packet class`) represents class (`Packet`).
- Class instance variable are instance variable of class
  -> should represent the state of class: number of created instances, number of messages sent, superclasses, subclasses....
- Class methods represent CLASS behavior: instance creation, **class** initialization, counting the number of instances....
- If you weaken the second point: class state and behavior can be used to define common properties shared by all the instances

  - Ex: If we want to encapsulate the way “no next node” is coded. Instead of writing:

```
aNode nextNode isNil not => aNode hasNextNode

Node>>hasNextNode
   ^ self nextNode = self noNextNode
Node>>noNextNode
   ^self class noNextNode
Node class>>noNextNode
   ^ nil
```

Class Initialization

- Automatically called by the system at **load time** or explicitly by the programmer.
  - Used to initialize a classVariable, a pool dictionary or class instance variables.
  - ‘`Classname initialize`’ at the end of the saved files.

```
Magnitude subclass: #Date
   instanceVariableNames: 'day year'
   classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames SecondsInDay WeekDayNames'
   poolDictionaries: ''
   category: 'Magnitude-General'

Date class>>initialize
    "Initialize class variables representing the names of the months and days and the number of
seconds, days in each month, and first day of each month. "
    MonthNames := #(January February March April May
    June July August September October November December ).
   SecondsInDay := 24 * 60 * 60.
   DaysInMonth := #(31 28 31 30 31 30 31 31 30 31 30 31 ).
   FirstDayOfMonth := #(1 32 60 91 121 152 182 213 244 274 305 335 ).
   WeekDayNames := #(Monday Tuesday Wednesday Thursday Friday Saturday Sunday )
```


A Case Study: Scanner

```
Scanner new
scanTokens: 'identifier keyword: 8r31 ''string'' embedded.period key:word: .   '
-> #(#identifier #keyword: 25 'string' 'embedded.period' #key:word: #'.')
```

- Class Definition

```
Object subclass: #Scanner
   instanceVariableNames: 'source mark prevEnd hereChar token tokenType saveCom-
ments currentComment buffer typeTable '
   classVariableNames: 'TypeTable '
   poolDictionaries: ''
   category: 'System-Compiler-Public Access'
```

- Why having an instance variable and a classVariable denoting the same object (the scanner table)?
  - TypeTable is used to initialize once the table
  - typeTable is used by every instance and each instance can customize the table (copying).

A Case Study: Scanner (II)

```
Scanner>>initialize
"Scanner initialize"
| newTable |
newTable := ScannerTable new: 255 withAll: #xDefault. "default"
newTable atAllSeparatorsPut: #xDelimiter.
newTable atAllDigitsPut: #xDigit.
newTable atAllLettersPut: #xLetter.
newTable at: $_ asInteger put: #xLetter.
'!%&*+,-/<=>?@\~' do: [:bin | newTable at: bin asInteger put: #xBinary].
"Other multi-character tokens"
newTable at: $" asInteger put: #xDoubleQuote.
...
"Single-character tokens"
newTable at: $# asInteger put: #literalQuote.
newTable at: $( asInteger put: #leftParenthesis.
...
newTable at: $^ asInteger put: #upArrow.  "spacing circumflex, formerly up arrow"
newTable at: $| asInteger put: #verticalBar.
TypeTable := newTable
```

A Case Study: Scanner (III)

- Instances only access the type table via the instance variable that points to the table that has been initialized once.

```
Scanner class>> new
   ^super new initScanner
Scanner>>initScanner
   buffer := WriteStream on: (String new: 40).
   saveComments := true.
   typeTable := TypeTable
```

- A subclass just has to specialize initScanner without copying the initialization of the table

```
MyScanner>>initScanner
   super initScanner
   typeTable := typeTable copy.
   typeTable at: $( asInteger put: #xDefault.
   typeTable at: $) asInteger put: #xDefault.
```


Why are Coupled Classes bad?

```
Packet>>addressee
   ^addressee
Workstation>>accept: aPacket
      aPacket addressee = self name
         ifTrue:[ Transcript show: 'A packet  is accepted by the Workstation ',
                             self name asString]
         ifFalse: [super accept: aPacket]
```

- If Packet changes the way addressee is represented, Workstation, Node, PrinterServer have to be changed too

The Law ot Demeter

- You should only send messages to:
  - an argument passed to you
  - an object you create
  - self, super
  - your class
- Avoid global variables
- Avoid objects returned from message sends other than self

```
someMethod: aParameter
   self foo.
   super someMethod: aParameter.
   self class foo.
   self instVarOne foo.
   instVarOne foo.
   self classVarOne foo.
   classVarOne foo.
   aParameter foo.
   thing := Thing new.
   thing foo
```

The Law of Demeter (II)

- Example

```
NodeManager>>declareNewNode: aNode
   |nodeDescription|
   (aNode isValid)                              "Ok passed as an argument to me"
      ifTrue: [ aNode certified].
   nodeDescription := NodeDescription for: aNode.
   nodeDescription localTime.                   "I created it"
   self addNodeDescription: nodeDescription.    "I can talk to myself"
   nodeDescription data                         "Wrong I should not know"
      at: self creatorKey                       "that data is a dictionary"
      put: self creator
```


About the Use of Accessors

- Literature says: “Access instance variables using methods”
  - Be consistent inside a class, do not mix direct access and accessor use
  - First think accessors as **private** methods that should **not** be invoked by clients
  - Only when necessary put accessors in accessing protocol

```
Scheduler>>initialize
   tasks := OrderedCollection new.

Scheduler>>tasks
   ^tasks
```

- BUT: accessors methods should be PRIVATE by default at least at the beginning
- Accessors are good for lazy initialization

```
Schedule>>tasks
   tasks isNil ifTrue: [task := ...].
   ^tasks
```


About the Use of Accessors (II)

- The fact that accessors are methods doesn’t provide you with a good data encapsulation.
- If they are mentionned as public (no enforcement in Smalltalk) you could be tempted to write in a client:

```
ScheduledView>>addTaskButton
   ...
   model tasks add: newTask
```

  - What’s happen if we change the representation of tasks? If tasks is now an array *it will break*
- Take care about the coupling between your objects and provide a good interface!

```
Schedule>>addTask: aTask
   tasks add: aTask
```

- Return consistenly the receiver or the element but not the collection (otherwise people can look inside and modify it) or return a copy of it.

About the Use of Accessors (III)

- Alan Knight: Never do the work somebody else can do!

```
XXX>>m
   total := 0.
   aPlant bilings do: [:each | (each status == #paid and: [each date>startDate])
                                 ifTrue: [total := total + each amount]].
```

- Instead write

```
XXX>m
   total := aPlant totalBillingsPaidSince: startDate
```


Provide a Complete Interface

```
Packet>>addressee
   ^addressee
Workstation>>accept: aPacket
      aPacket addressee = self name
         ifTrue:[ Transcript show: 'A packet  is accepted by the Workstation ',
                                 self name asString]
         ifFalse: [super accept: aPacket]
```

- It is the responsibility of an object to propose a complete interface that propects itself from client intrusion.
- Shift the responsibility to the Packet object

```
Packet>>isAddressedTo: aNode
   ^ addressee = aNode name
Workstation>>accept: aPacket
      (aPacket isAddressedTo: self)
         ifTrue:[ Transcript show: 'A packet  is accepted by the Workstation ',
self name asString]
         ifFalse: [super accept: aPacket]
```


Factoring Out Constants

- Ex: We want to encapsulate the way “no next node” is coded. Instead of writing:

```
Node>>nextNode
   ^ nextNode
NodeClient>>transmitTo: aNode
   aNode nextNode = ‘no next node’
      ...
```

- Write:

```
NodeClient>>transmitTo: aNode
   aNode hasNextNode
      ....
Node>>hasNextNode
   ^ (self nextNode = self class noNextNode) not

Node class>>noNextNode
   ^ ‘no next node’
```

Initializing without Duplicating

```
Node>>initialize
    accessType := 'local'
    ...
Node>>isLocal
    ^ accessType = 'local'
```

- It's better to write

```
Node>>initialize
    accessType := self localAccessType

Node>>isLocal
    ^ accessType = self localAccessType

Node>>localAccessType
    ^ 'local'
```

- Ideally you could be able to change the constant without having any problems.
- You may have to have mapping tables from model constants to UI constants or database constants.

Constants Needed at Creation Time

- Works well for:

```
Node class>>localNodeNamed: aString
    |inst|
    inst := self new.
    inst name: aString.
    inst type: inst localAccessType
```

- If you want to have the following creation interface

```
Node class>>name: aString accessType: aType
    ^self new name: aString ; accessType: aType
Node class>>name: aString
    ^self name: aString accessType: self localAccessType
```

- You need:

```
Node class>>localAccessType
    ^ 'local'
```

- -> Factor the constant between class and instance level

```
Node>>localAccessType
    ^self class localAccessType
```

- -> You could also use a ClassVariable that is shared between a class and its instances.

Type Checking for Dispatching

- How to invoke a method depending on the receiver and an argument?
  - A not so good solution:

```
PSPrinter>>print: aDocument
    ^ aDocument isPS
        ifTrue: [self printFromPS: aDocument]
        ifFalse: [self printFromPS: aDocument asPS]

PSPrinter>>printFormPS: aPSDoc
    <primitive>

PdfPrinter>>print: aDocument
    ^ aDocument isPS
        ifTrue: [self printFromPDF: aDocument asPDF]
        ifFalse: [self printFromPDF: aDocument]

PdfPrinter>>printFormPS: aPdfDoc
    <primitive>
```

- As we do not know how to coerce form the PSPrinter to a PdfPrinter we only use coercion between documents.

Double Dispatch

- How to invoke a method depending on the receiver and an argument?
- Solution: use the information given by the single dispatch and redispatch with the argument (send a message back to the argument passing the receiver as an argument)

```
(a)   PSPrinter>>print: aDoc
          ^ aDoc printOnPSPrinter: self
(b)   PdfPrinter>>print: aDoc
          ^ aDoc printOnPdfPrinter: self

(c) PSDoc>>printOnPSPrinter: aPSPrinter
        <primitive>
(d) PdfDoc>>printOnPdfPrinter: aPSPrinter
        aPSprinter print: self asPS

(e) PSDoc>>printOnPSPrinter: aPdfPrinter
        aPdfPrinter print: self asPdf
(f) PdfDoc>>printOnPdfPrinter:aPdfPrinter
        <primitive>
```

- Some Tests:

```
psptr print: psdoc =>(a->c)
pdfptr print: pdfdoc => (b->f)
psptr print: pdfdoc => (a->d->b->f)
pdfptr print: psdoc => (b->e->b->f)
```


A Step Back

- Example: Coercion between Float and Integer
  - Not a really good solution:

```
Integer>>+ aNumber
       (aNumber isKindOf: Float)
          ifTrue: [ aNumber asFloat + self]
          ifFalse: [ self addPrimitive: aNumber]

Float>>+ aNumber
       (aNumber isKindOf: Integer)
          ifTrue: [aNumber asFloat + self]
          ifFalse: [self addPrimitive: aNumber]
```

- Here receiver and argument are the same, we can coerce in both senses.

Double Dispatch (II)

```
(a)   Integer>>+ aNumber
          ^ aNumber sumFromInteger: self
(b)   Float>>+ aNumber
          ^ aNumber sumFromFloat: self

(c) Integer>>sumFromInteger: anInteger
        <primitive: 40>
(d) Float>>sumFromInteger: anInteger
        ^ anInteger asFloat + self

(e) Integer>>sumFromFloat: aFloat
        ^aFloat + self asFloat
(f) Float>>sumFromFloat: aFloat
        <primitive: 41>
```

- Some Tests:

```
1 + 1: (a->c)
1.0 + 1.0: (b->f)
1 + 1.0: (a->d->b->f)
1.0 + 1: (b->e->b->f)
```

Methods are the Basic Units of Reuse

```
Node>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
   self window add:
      UINode new with:
         (self bandWidth * averageRatio / defaultWindowSize)
   ...
```

- We are forced to copy the method!

```
SpecialNode>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate + minimalRatio / maximiseViewRa-
tio.
   self window add:
      UINode new with: (self bandWidth * averageRatio / defaultWindowSize)
   ...
```


Methods are the Basic Units of Reuse (II)

- Self sends = planning for Reuse

```
Node>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self defaultNodeSize.
   self window add:
      UINode new with:
         (self bandWidth * averageRatio / defaultWindowSize)
   ...
   Node>>defaultNodeSize
      ^self mainWindowCoordinate / maximiseViewRatio

   SpecialNode>>defaultNodeSize
      ^self mainWindowCoordinate + minimalRatio / maximiseViewRatio
```


Methods are the Basic Units of Reuse (III)

```
Node>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
   self window add:
      UINode new with:
         (self bandWidth * averageRatio / defaultWindowSize).
   ...
```

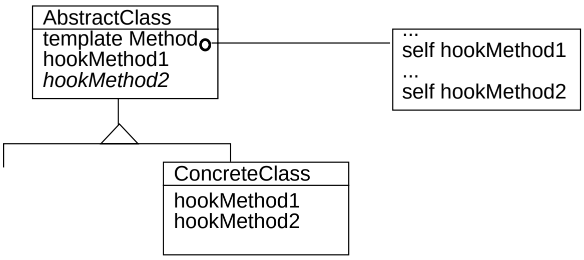
- We are forced to copy the method!

```
SpecialNode>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
   self window add:
      ExtendedUINode new with:
         (self bandWidth * averageRatio / defaultWindowSize).
```

Class Factories

```
Node>>computeRatioForDisplay
   |averageRatio |
   averageRatio := 55.
   self window add:
      self UIClass new with:
         (self bandWidth * averageRatio / self defaultWindowSize)
   ...

   Node>>UIClass
      ^UINode

   SpecialNode>>UIClass
      ^ExtendedUINode
```


Hook and Template Methods



- Hook methods do not have to be abstract, they may define default behavior or no behavior at all.
- This has an influence on the instantiability of the superclass.

Hook Example: Copying

```
Object>>copy
   " Answer another instance just like the receiver. Subclasses normally override the postCopy
message, but some objects that should not be copied override copy. "

   ^self shallowCopy postCopy

Object>>shallowCopy
   "Answer a copy of the receiver which shares the receiver's instance
   variables."

   <primitive: 532>
   ....

Object>>postCopy
   " Finish doing whatever is required, beyond a shallowCopy, to implement 'copy'.
   Answer the receiver. This message is only intended to be sent to the newly created instance.
   Subclasses may add functionality, but they should always do super postCopy first. "
   " Note that any subclass that 'mixes in Modelness' (i.e., implements dependents with an instance
   variable) must include the equivalent of 'self breakDependents'"

   ^self
```

Hook Specialisation

```
Bag>>postCopy
    "Make sure to copy the contents fully."

    | new |
    super postCopy.
    new := contents class new: contents capacity.
    contents keysAndValuesDo:
        [:obj :count | new at: obj put: count].
    contents := new.
```


Hook and Template Example: Printing

```
Object>>printString
    "Answer a String whose characters are a description of the receiver."

    | aStream |
    aStream := WriteStream on: (String new: 16).
    self printOn: aStream.
    ^aStream contents

Object>>printOn: aStream
    "Append to the argument aStream a sequence of characters
    that describes the receiver."

    | title |
    title := self class name.
    aStream nextPutAll:
        ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
    aStream print: self class
```


Override of the Hook

```
Array>>printOn: aStream
    "Append to the argument, aStream, the elements of the Array
    enclosed by parentheses."

    | tooMany |
    tooMany := aStream position + self maxPrint.
    aStream nextPutAll: '#('.
    self do: [:element |
            aStream position > tooMany
                ifTrue:
                    [aStream nextPutAll: '...(more)...)'.
                    ^self].
            element printOn: aStream]
        separatedBy: [aStream space].
    aStream nextPut: $)

False>>printOn: aStream
    "Print false."

    aStream nextPutAll: 'false'
```

Specialization of the Hook

- The class Behavior that represents a class extends the default hook but still invokes the default one.

```
Behavior>>printOn: aStream
    "Append to the argument aStream a statement of which
    superclass the receiver descends from."

    aStream nextPutAll: 'a descendent of '.
    superclass printOn: aStream
```


Behavior Up and State Down

- 4 steps

1. Define classes by behavior, not state
2. Implement behavior with abstract state: if you need state do it indirectly via messages. Do not reference the state variables directly
3. Identify message layers: implement class's behavior through a small set of kernel method
4. Defer identification of state variable: The abstract state messages become kernel methods that require state variables. Declare the variable in the subclass and defer the kernel methods' implementation to the subclasses

```
Collection>>removeAll: aCollection
    aCollection do: [:each | self remove: each]
    ^ aCollection

Collection>>remove: oldObject
    self remove: oldObject ifAbsent: [self notFoundError]

Collection>>remove: anObject ifAbsent: anExceptionBlock
    self subclassResponsibility
```


Guidelines for Creating Template Methods

- Simple implementation. Implement all the code in one method.
- Break into steps. Comment logical subparts
- Make step methods. Extract subparts as methods
- Call the step methods (including when using the refactoring browser)
- Make constant methods, i.e., methods doing nothing else than returning.
- Repeat steps 1-5 if necessary on the methods created

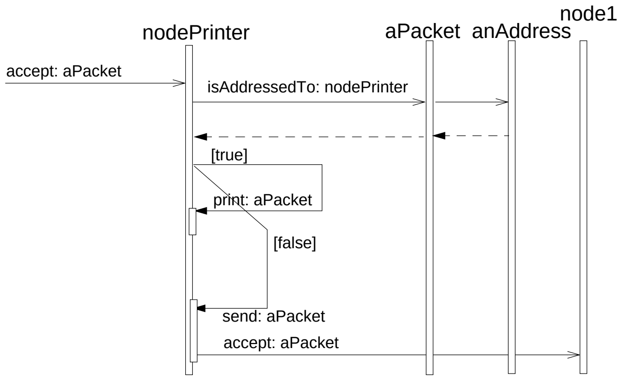
Towards Delegation: Matching Addresses

- **New requirement:** A document can be printed on different printers for example lw100s or lw200s depending on which printer is first encountered.
  - -> Packet need more than one destination
- **Ad-hoc Solution:**

```
LanPrinter>>accept: aPacket
   (thePacket addressee = #*lw*)
      ifTrue: [ self print: thePacket]
      ifFalse: [ (thePacket isAddressedTo: self)
                 ifTrue: [self print: thePacket]
                 ifFalse: [super accept: thePacket]]
LanPrinter>>print: aPacket
   Transcript
      show: self name ;
      '***** printing *****';cr
      show: aPacket contents ;cr
```

- **Limits:**
  - not general
  - brittle because based on a convention
  - adding a new kind of address behavior requires editing the class Printer

Reify and Delegate



- An alternative solution: isAddressedTo: could be sent directly to the address
- With the current solution, the packet can still control the process if needed

Reifying Address

Reify: v. making something an object (philosophy)

- NodeAddress is responsible for identifying the packet receivers

```
Object subclass: #NodeAddress
   instanceVariableNames: 'id'

NodeAddress>>isAddressedTo: aNodeAddress
   ^ self id = aNodeAddress id

Packet>>isAddressedTo: aNode
   ^ self addressee isAddressedTo: aNode name
```

- Having the same name for packet and for address is not necessary but the name is meaningful!
  - Refactoring Remark: name was not a good name anyway, and now it has become an address -> we should rename it.

Matching Address

```
Address subclass: #MatchingAddress
   instanceVariableNames: ‘’

NodeAddress>>isAddressedTo: aNodeAddress
   ^ self id match: aNodeAddress id
```

- Works for packets with matchable addresses
  `Packet send: ‘lulu’ to: (MatchingAddress with: #*lw*)`
- Does not work for nodes with matchable addresses because the match is directed. But it corresponds to the requirements!
  `Node withName: (MatchingAddress with: #*lw*)`

```
Packet>>isAddressedTo: aNode
   ^ self addressee isAddressedTo: aNode name
```

- Remarks
  - inheritance class relationship is not really good because we can avoid duplication (coming soon)
  - Creation interfaces could be drastically improved

Addresses

```
Object subclass: #Address
   instanceVariableNames: ‘id‘

Address>>isAddressedTo: anAddress
   ^self subclassResponsibility

Address subclass: #NodeAddress
   instanceVariableNames: ‘‘

Address subclass: #MatchingAddress
   instanceVariableNames: ‘‘
```


Trade-Off

- Delegation Pros
  - No blob class: one class one responsibility
  - Variation possibility
  - Pluggable behavior without inheritance extension
  - Runtime pluggability

- Delegation Cons
  - Difficult to follow responsibilities and message flow
  - Adding new classes = adding complexities (more names)
  - New object

Designing Classes for Reuse

- Encapsulation principle: minimize data representation dependencies
  - Complete interface
  - No overuse of accessors
  - Responsibility of the instance creation
- Loose coupling between classes
- Methods are units of reuse (self send)
- Use polymorphism as much as possible to avoid type checking
- Behavior up and state down
- Use correct names for class
- Use correct names for methods

Do not overuse conversions

```
nodes asSet
```

- removes all the duplicated nodes (if node knows how to compare). But a systematic use of asSet to protect yourself from duplicate is not good

```
nodes asSet asOrderedCollection
```

- returns an ordered collection after removing duplicates
- Look for the real source of duplication if you do not want it!

Hiding missing information

```
Dictionary>>at: aKey
```

- This raises an error if the key is not found

```
Dictionary>>at: aKey ifAbsent: aBlock
```

- This allows one to specify action <aBlock> to be done when the key does not exist. Do not overuse it:

```
nodes at: nodeId ifAbsent:[]
```

- This is bad because at least we should know that the `nodeId` was missing

Different Self/Super

- Do not invoke a super with a different method selector. It's bad style because it links a class and a superclass. This is dangerous in case the software evolves.

```
Packet class>>new
   self error: 'Packet should only be created using send:to:'

Packet class>>send: aString to: anAddress
   ^ super new contents: aString ; addressee: anAddress
```

- **Use** `basicNew` and `basicNew:`

```
Packet class>>send: aString to: anAddress
   ^ self basicNew contents: aString ; addressee: anAddress
```

- Never override `basicNew` and `basicNew:` (another name allocate only create instance without instance variable initialization)

22. Selected Idioms

- The Object Manifesto: Be lazy and be private
  - Never do the job that you can delegate to another one
  - Never let someone else plays with your private data
- The Programmer Manifesto: Say something only once

Composed Methods

- How do you divide a program into methods?
  - Messages take time
  - Flow of control is difficult with small methods
- But:
  - Reading is improved
  - Performance tuning is simpler (Cache...)
  - Easier to maintain / inheritance impact

Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction.

```
Controller>>controlActvity
   self controlInitialize.
   self controlLoop.
   self controlTerminate
```


Constructor Method

- How do you represent instance creation?
  - Most simple way: `Packet new addressee: # mac ; contents: ‘hello mac’`
- Good if there are different combinations of parameters. But you have to read the code to understand how to create an instance.
- Alternative: make sure that there is a method to represent each valid way to create an instance.

Provide methods in class “instance creation” protocol that create well-formed instances. Pass all required parameters to them

```
Packet class>>send: aString to: anAddress
   ^ self basicNew contents: aString ; addressee: anAdress ; yourself
Point class>>x:y:
Point class>> r: radiusNumber theta: thetaNumber
   ^ self
      x: radiusNumber * thetaNumber cos
      y: radiusNumber * thetaNumber sin
SortedCollection class>>sortBlock: aBlock
```

Constructor Parameter Method

- Once you define a constructor with paramteres, how do you pass them to the newly created instance?

```
Packet class>>send: aString to: anAddress
    ^ self basicNew contents: aString ; addressee: anAdress ; yourself
```

- But this violates the “say things once and only once” rule (initialize)

Code a single method in the “private” procotol that sets all the variables. Preface its name with “set”, then the names of the variables.

```
Packet class>>send: aString to: anAddress
    ^ self basicNew setContents: aString addressee: anAddress
Packet>>setContents: aString addressee: anAddress
    contents:= aString.
    addressee := anAddress.
    ^self
```

Note `self` (Interesting Result) in `setContents:addressee`, because the return value of the method will be used as the return of the caller

Query Method

- How do you represent testing a property of an object?
- What to return from a method that tests a property?
- Instead of:

```
Switch>>makeOn
    status := #on
Switch>>makeOff
    status := #off
Switch>>status
    ^status
Client>>update
    self switch status = #on ifTrue: [self light makeOn]
    self switch status = #off ifTrue: [self light makeOff]
```

- It is better to define
  `Switch>>isOn, Switch>>isOff`

Provide a method that returns a Boolean in the “testing” protocol. Name it by prefacing the property name with a form of “`be`” or “`has`”- is, was, will, has

Switch>>on is not a good name... #on: or #isOn ?

Boolean Property Setting Method

- How do you set a boolean property?

```
Switch>>on: aBoolean
    isOn := aBoolean
```

- Expose the representation of the status to the clients
- Responsibility of who turn off/on the switch: the client and not the object itself

Create two methods beginning with “be”. One has the property name, the other the negation. Add “toggle” if the client doesn’t want to know about the current state

```
beVisible/beInvisible/toggleVisible
```

Comparing Method

- How do we order objects?
- <,<=,>,>= are defined on Magnitude and its subclasses.

Implement "<=" in "comparing" protocol to return true if the receiver should be ordered before the argument

- We can also use `sortBlock:` of `SortedCollection class`

```
...sortBlock: [:a :b | a income > b income]
```


Execute Around Method

- How do we represent pairs of actions that have to be taken together?
  - When a filed is opened it has to be closed....
- Basic solutions: under the client responsibility, he should invoke them on the right order.

Code a method that takes a Block as an argument. Name the method by appending "During: aBlock" to the name of the first method that have to be invoked. In the body of the Execute Around Method, invoke the first method, evaluate the block, then invoke the second method.

```
File>>openDuring: aBlock
    self open.
    aBlock value.
    self close
```

```
File>>openDuring: aBlock
    self open.
    [aBlock value]
        valueNowOrUnwindDo: [self close]
```

```
Cursor>>showWhile: aBlock
    | oldcursor |
    oldcursor := self class currentCursor.
    self show.
    ^aBlock valueNowOrOnUnwindDo:
           [oldcursor show]
```


Choosing Message

- How do you execute one of several alternatives?

```
responsible := (anEntry isKindOf: Film)
    ifTrue:[anEntry producer]
    ifFalse:[anEntry author]
```

- Use polymorphism

```
Film>>responsible
    ^self producer
Entry>>responsible
    ^self author
responsible := anEntry responsible
```

Send a message to one of several different of objects, each of which executes one alternative

- Examples:

```
Number>>+ aNumber
Object>>printOn: aStream
Collection>>includes:
```

- A Choosing Message can be sent to self in anticipation of future refinement by inheritance. See also the State Pattern.

Intention Revealing Message

- How do you communicate your intent when the implementation is simple?
- We are not writing for computer but for reader

```
ParagraphEditor>>highlight: aRectangle
    self reverse: aRectangle
```

- If you would replace `#highlight:` by `#reverse:` , the system will run in the same way but you would reveal the implementation of the method.

Send a message to self. Name the message so it communicates what is to be done rather than how it is to be done. Code a simple method for the message.

```
Collection>>isEmpty
    ^self size = 0

Number>>reciprocal
    ^ 1 / self
```


Intention Revealing Selector

- How do you name a method?
  - If we choose to name after HOW it accomplished its task

```
Array>>linearSearchFor:
Set>>hashedSearchFor:
BTree>>treeSearchFor:
```

  - These names are not good because you have to know the type of the objects.

Name methods after WHAT they accomplish

- Better:

```
Collection>>searchFor:
```

- Even better:

```
Collection>>includes:
```

  - Try to see if the name of the selector would be the same in a different implementation.

Name your Methods Well

- Not precise, not good

```
setType: aVal
    "compute and store the variable type"
    self addTypeList: (ArrayType with: aVal).
    currentType :=  (currentType computeTypes: (ArrayType with: aVal))
```

- Precise, give to the reader a good idea of the functionality and not about the implementation

```
computeAndStoreType: aVal
    "compute and store the variable type"
    self addTypeList: (ArrayType with: aVal).
    currentType :=  (currentType computeTypes: (ArrayType with: aVal))
```

- Instead Of:

```
setTypeList: aList
    "add the aList elt to the Set of type taken by the variable"
    typeList add: aList.
```

- Write:

```
addTypeList: aList
    "add the aList elt to the Set of type taken by the variable"
    typeList add: aList.
```

do: / collect:

- Instead of writing:

```
|index|
index := 1.
[index <= aCollection size] whileTrue:
    [... aCollection at: index...
    index := index + 1]
```

- Write:

```
aCollection do: [:each | ...each ...]
```

- Instead of writing:

```
absolute: aCollection
|result|
result := aCollection species new: aCollection size.
1 to: aCollection size do: [ :each | result at: each put: (aCollection at: each) abs].
^ result
```

- Write:

```
absolute: aCollection
    ^ aCollection collect: [:each| each abs]
```

  - Note that this solution works well for indexable collection and also for sets. The previous doesn't.

isEmpty / includes:

- Instead of writing:

```
...aCollection size = 0 ifTrue: [...]
...aCollection size > 0 ifTrue: [...]
```

- Write:

```
... aCollection isEmpty
```

- Instead of writing:

```
|found|
found := false.
aCollection do: [:each| each = anObject ifTrue: [found : = true]].

|found|
found := (aCollection detect: [:each| each | anObject] ifNone:[ nil]) notNil.
```

- Write:

```
|found|
found := aCollection includes: anObject
```


Naming Suggestions

- Attributes: The type of an attribute should not be reflected in its name

  `nodes`

  instead of

  `nodeArray`

- Classes:
  - Name a superclass with a single word that conveys its purpose in the design

```
Number
Collection
View
Model
```

  - Name subclasses in your hierarchy by prepending an adjective to the superclass name

```
OrderedCollection
SortedCollection
LargeInteger
```

Reversing Method

- How to code a smooth flow of messages?

```
Point>>printOn: aStream
    x printOn: aStream
    aStream nextPutAll: ' @'.
    y printOn: aStream
```

- Here three objects receive different messages.

Code a method on the parameter. Derive its name form the original message. Take the original receiver as a parameter to the new method. Implement the method by sending the original message to the original receiver.

- But creating new selectors just ofr fun is not a good idea. Each selector must justify its existence.

```
Stream>>print: anObject
    anObject printOn: self
Point>>printOn: aStream
    aStream print: x; nextPutAll: ' @'; print: y
```

  - Note that the receiver can now change without affecting the other parameters

Debug Printing Method

- How do you code the default printing method?
- There are two audiences:
  - you (a lot of information)
  - your clients (should not be aware of the internal)

Override printOn: to provide information about object's structure to the programmer

- In VisualWorks, two needs are supported
  - `displayString` for clients
  - `printString` for you (call printOn:)

Method Comment

- How do you comment methods?
- Templates are not a good idea. Uses:
  - Intention Revealing Selector says what the method does
  - Type Suggesting Parameter Name says what the arguments are expected to be.....

Communicate important information that is not obvious from the code in a comment at the beginning of the method

- Example of important information:
  - Method dependencies, preconditions
  - To do
  - Reasons for changes (in a base class)

```
(self flags bitAnd: 2r1000) = 1 "Am I visible?"
    ifTrue:[...]

isVisible
^(self flags bitAnd: 2r1000) = 1

self isVisible ifTrue:[...]
```

Delegation

- How does an object share implementation whitout inheritance?
- With inheritance
  - code in written in context of superclasses
  - in rich hierarchies, you may to read and understand many superclasses
  - how to simulate multiple inheritance (if this is really necessary)

Pass part of its work on to another object

- Many objects need to display, all objects delegate to a brush-like object (Pen in VisualSmalltalk, GraphicsContext in VisualAge and VisualWorks)
- All the detailed code is concentrated in a single class and the rest of the system has a simplified view of the displaying.

Simple Delegation

- How do you invoke a disinterested delegate?
- Some important question on delegation:
  - is the identity of the delegating object important? The delegating object can pass itself to be notified by the delegate.The delegate could not want to have an explicit reference to the delegating but still need access to it.
  - is the state of the delegating object important to the delegate? If the delegate has no reason to need the identity of the delegating object and it is self-contained to accomplish its task without additional state: Simple Delegation

Delegate messages unchanged

- Suppose an object that acts a LITTLE as a collection but has lots of other protocols, instead fo inheriting from a collection, delegates to a collection.
  - Collection doesn't care who invoked it. No state from the delegating is required.

Self Delegation

- How do you implement delegation to an object that needs reference to the delegating object?
- One way is to have a reference in the delegate to the delegating.
- Drawbacks:
  - extra complexity,
  - each time the delegate changes, one should destroy the old reference and set a new
  - each delegate can only be used by one delegating,
  - If creating multiple copies of the delegate is expensive or impossible, this does not work

Pass along the delegating object (i.e., self) in an additional parameter called "for:"

Self Delegation - Example

- In VisualSmalltalk, hashed collections (dictionaries) use a hash table. Variants of the hash table can be used depending on different criterias.
- Hash value is implemented differently by different collections. Dictionaries compute hash by sending “hash” and IdentityDictionaries by sending “basicHash”

```
Dictionary>>at: key put: value
    self hashTable at: key put: value for: self

HashTable>>at: key put: value for: aCollection
    |hash|
    hash := aCollection hashOf: key
    ...
Dictionary>>hashOf: anObject
    ^anObject hash
IdentityDictionary>>hashOf: anObject
    ^anObject basicHash
```

- The hierarchy of hashed Collections is then independent of the hierarchy of the HashTable

Pluggable Behavior

- How do you parameterize the behavior of an object?
- In the class based model instances have private values and share the behavior.When you want a different behavior you create a new class. But creating class is not always valuable: imagine a large number of classes with only a single method.
- Questions to consider: how much felxibility you need? How many methods will need to vary dynamically? How hard is it to follow the code? Will clients specify the code to be plugged?

Add a variable that will be used to trigger different behavior

- Typical examples are user-interface object that have to display the contents of many dfferent objects

Pluggable Selector

- How do you code simple instance specific behavior?
- The simplest way to implement Pluggable Behavior is to store a selector.

Add a variable that contains a selector to be performed. Append “Message” to the Role Suggesting Instance Variable Name. Create a Composed Method that simply performs the selector.

```
ListPane>>printElement: anObject
    ^anObject printString
```

- And subclasses only specializing

```
DollarListPane>>printElement: anObject
    ^anObject asDollarFormatString
DescriptionListPane>>printElement: anObject
    ^ anObject description
ListPane>>printElement: anObject
    ^anObject perform: printMessage
ListPane>>initialize
    printMessage := #printString
```

- Readibility: harder to follow than simple class-based behavior
- Extent: if you need more than twice per object use State Object

Pluggable Block

- How do you code COMPLEX Pluggable Behavior that is not quite worth its own class?

Add an instance variable to store a Block. Append “Block” to the Role Suggesting Instance Variable Name. Create aComposed Method to evaluate the Block to invoke the Pluggable Behavior.

- Drawbacks: Enormous cost, readibility is worse, blocks are difficult to store
- PluggableAdaptor in VisualWorks allows one to map any interface to the value model. A simplified version:

```
Car>>speedAdaptor
    ^PluggableAdaptor
    getBlock: [self speed]
    putBlock: [:newSpeed| self speed: newSpeed]
PluggableAdaptor>>value
    ^getBlock value
PluggableAdaptor>>value: anObject
    putBlock value: anObject
```

23. Selected Design Patterns

- Singleton
- Composite
- Null Object

The Singleton Pattern

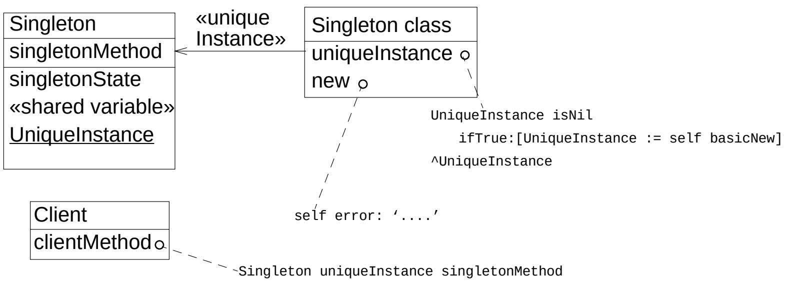
- **Problem:** We want a class with a unique instance.
- **Solution:** We specialize the `#new` class method so that if one instance already exists this will be the only one. When the first instance is created, we store and return it as result of `#new`.

```
|aLan|
aLan := NetworkManager new
aLan == LAN new -> true
aLan uniqueInstance == NetworkManager new -> true

NetWorkManager class
   instanceVariableNames: 'uniqueInstance '
NetworkManager class>>new
   self error: 'should use uniqueInstance'
NetworkManager class>>uniqueInstance
   uniqueInstance isNil ifTrue: [ uniqueInstance := self basicNew initialize].
   ^uniqueInstance
```

- Providing access to the unique instance is not always necesssary. It depends on what we want to express. The difference between `#new` and `#uniqueInstance` is that `#new` potentially initializes a new instance, while `#uniqueInstance` only returns the unique instance (there is no initialization).

Singleton (II) - Theory

- **Intent**: Ensure that a class has only one instance, and provide a global point of access to it
- **A Possible Structure**



Singleton (III) - Implementation

- In some Smalltalk dialects, singletons are accessed via a global variable (ex: NotificationManager uniqueInstance notifier).

```
SessionModel>>startupWindowSystem
    "Private - Perform OS window system startup"
    |oldWindows|
    ...
    Notifier initializeWindowHandles.
    ...
    oldWindows := Notifier windows.
    Notifier initialize.
    ...
    ^oldWindows
```

- Global Variable or Class Method Access
- Global Variable Access is dangerous: if we reassign Notifier we lose all references to the current window.
- Class Method Access is better because it provides a single access point. This class is responsible for the singleton instance (creation, initialization,...).

Singleton (IV) - Implementation

- Singleton Variations
  - Persistent Singleton: only one instance exists and its identity does not change (ex: NotifierManager in Visual Smalltalk)
  - Transient Singleton: only one instance exists at any time, but that instance changes (ex: SessionModel in Visual Smalltalk, SourceFileManager, Screen in VisualWorks)
  - Single Active Instance Singleton: a single instance is active at any point in time, but other dormant instances may also exist. Project in VisualWorks, ControllerManager.
- In Smalltalk we cannot prevent a client to send a message (protected in C++). To prevent additional creation we can redefine new/new:

```
Object subclass: #Singleton
    instanceVariableNames: ''
    classVariableNames: 'UniqueInstance'
    poolDictionaries: ''

Singleton class>>new
    self error: 'Class ', self name, ' cannot create new instances'
```


Singleton (V) - Implementation

- Providing Access:
  - Lazy Access, however with this solution we lose the initialization part of the superclass

    ```
    Singleton class>>uniqueInstance
        UniqueInstance isNil ifTrue:[UniqueInstance := self basicNew].
        ^UniqueInstance
    ```

  - Wan also try the following, if the initialization was done using initialize

    ```
        ... ifTrue: [UniqueInstance := self basicNew initialize] ...
    ```

  - The following is also done, but is bad practice and may break

    ```
        ... ifTrue: [UniqueInstance := super new] ...
    ```

- Access using new

```
Singleton class>>new
    ^self uniqueInstance
```

  - The intent (uniqueness) is not clear anymore! New is normally used to return newly created instances. The programmer does not expect this:

```
|screen1 screen2|
screen1 := Screen new.
screen2 := Screen uniqueInstance
```

Singleton (VI) - Implementation

- Singleton for an entire subhierarchy of classes:

```
Object subclass: #Singleton
    instanceVariableNames: ‘’
    classVariableNames: ‘UniqueInstance’
```

- ClassVariables are shared by all the subclasses
- Singleton for each of the classes in an hierarchy

```
Singleton class instanceVariableNames: ‘uniqueInstance’
Singleton class>>uniqueInstance
    uniqueInstance isNil ifTrue:[uniqueInstance := self basicNew].
    ^uniqueInstance
```

  - Instances variables of classes are private to the class
- When a class should only have one instance, it could be tempting to define all its behavior at the class level. But this is not good:
  - Class behavior represents behavior of classes: “Ordinary objects are used to model the real world. MetaObjects describe these ordinary objects”
  - Do not mess up this separation and do not mix domain objects with metaconcerns.
  - What’s happens if later on an object can have multiple instances? You would have to change a lot of client code!

The Composite Pattern

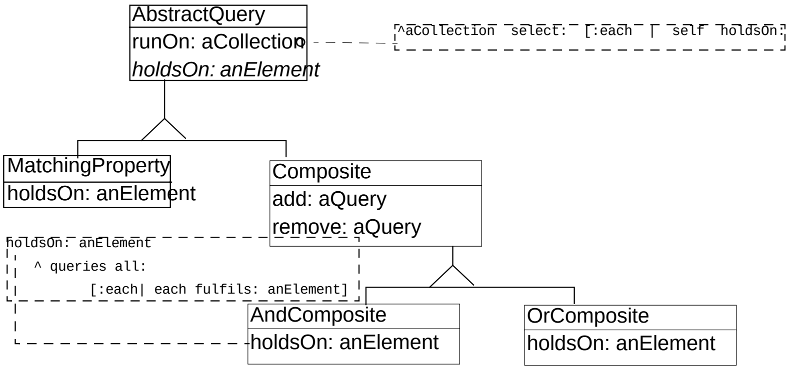
- A Case study: Queries. We want to be able to
  - Specify different queries over a repository

```
q1 := PropertyQuery property: #HNL with: #< value: 4.
q2 := PropertyQuery property: #NOM with: #> value: 10.
q3 := MatchName match: ‘*figure*’
```

  - Compose these queries and treat composite queries as one query

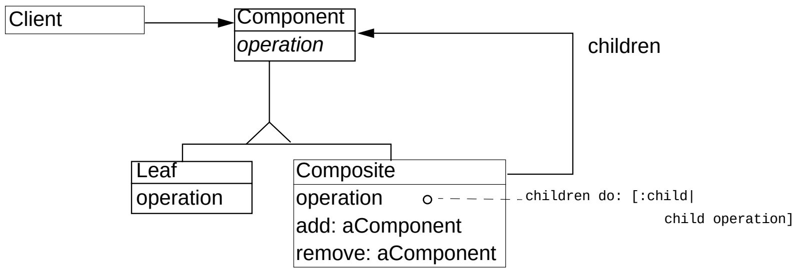
```
(e1 e2 e3 e4 ... en)((q1 and q2 and q4) or q3) -> (e2 e5)
composer := AndComposeQuery with: (Array with: q1 with: q2 with: q3)
```


Composite (II) - A Possible Solution



Composite (III) - Theory

- **Intent:** Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



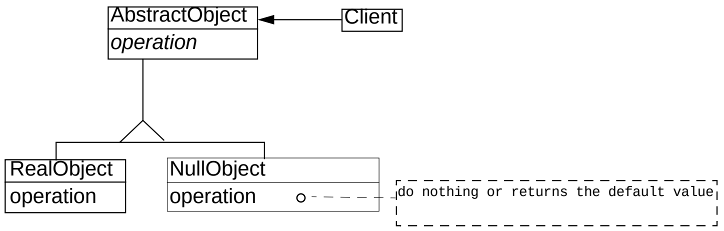
  - Composite not only groups leaves but can also contain composites
  - In Smalltalk add:, remove: do not need to be declared into Component but only on Composite. This way we avoid to have to define dummy behavior for Leaf

Composite (IV) - Implementation

- Use a Component superclass (To define the interface and factor code there)
- Consider implementing abstract Composite and Leaf (in case of complex hierarchy)
- Only Composite delegates to children
- Composites can be nested
- Composite sets the parent back-pointer (add:/remove:)
- Can Composite contain any type of child? (domain issues)
- Is the Composite’s number of children limited?
- Forward
  - Simple forward. Send the message to all the children and merge the resuslts without performing any other behavior
  - Selective forward. Conditionally forward to some children
  - Extended forward. Extra behavior
  - Override. Instead of delegating

The NullObject Pattern

- **Intent:** Provides a surrogate for another object that shares the same interface but does nothing. The NullObject encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators



NullObject (II) - With or Without

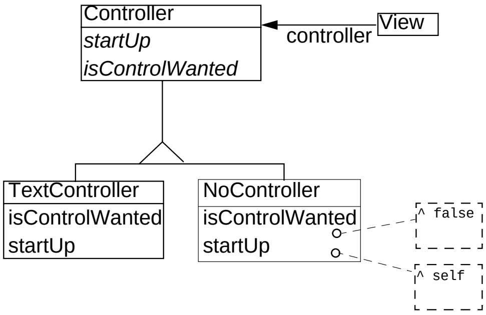
- Without this pattern, for example in MVC the View has to check that its controller is not nil before invoking the normal behavior.

```
VisualPart>>objectWantingControl
    ... ^ ctrl isNil ifFalse: [ctrl isControlWanted ifTrue: [self] ifFalse: [nil]]
```

- With NullObject, we avoid to make explicit tests

```
VisualPart>>objectWantingControl
    ... ^ ctrl isControlWanted ifTrue: [self] ifFalse: [nil]
Controller>>isControlActive
    ^self viewHasCursor and:[...]
Controller>>startUp
    self controlInitialize. self controlLoop. self controlTerminate
Controller>>isControlWanted
    ^self viewHasCursor
NoController>>isControlWanted
    ^false
NoController>>startUp
    ^self
NoController>>isControlActive
    ^false
```


NullObject (III) - Controller Hierarchy Example



NullObject (IV) - Consequences

- Advantages
  - Uses polymorphic classes: NullObject and real ones share the same interface so are interchangeable
  - Simplifies client code: Clients does not have to handle null case
  - Encapsulates do-nothing behavior: easy to identify, coded efficiently
  - Make do-nothing behavior reusable
- Disadvantages
  - Forces encapsulation: the same null object cannot be added to several classes unless they all delegate to a collaborator that can be a null object.
  - May cause class explosion: one class -> superclass and null object
  - Is non-mutable: a null object does not transform into a real object

NullObject (V) - Applicability

- Apply NullObject
  - When an object requires a collaborator that already exists before the NullObject pattern.
  - When some instances should do nothing
  - When you want clients to be able to ignore the difference between collaborators
  - When you want the do-nothing behavior
  - When all the do-nothing behavior is encapsulated in the collaborator class

- Do not apply NullObject (i.e., use a variable set to nil)
  - When very little code actually uses the variable directly
  - When the code that does use the variable is well encapsulated in one place
  - When the code that uses the variable handles it always the same way

NullObject (VI) - VisualWorks Examples

- **Null Strategies:** NoController in the (MVC) Controller hierarchy. NoController represents a controller that never wants control. It is the controller for views that is noninteractive.
  - DragMode implements the dragging of widgets in the window painter. SelectionDragMode allows the move of the widget, CornerDragMode lets the user resize it. NullDragMode responds to the mouse's drag motions by doing nothing.
- **Null Adapters:** NullInputManager in the InputManager hierarchy. An InputManager is a platform neutral object interface to platform events that affect internationalised input. Subclasses represent specific platforms. NullInputManager represents platforms that don't support internatialisation.
- **Reusable Nulls:** A NameScope represents a name scope -- static (global / pool / class pool), instance variables (of a class or class hierarchy), or local (argument / temporary, of a method or block). A StaticScope holds global and class variables, LocalScopes holds instance and temporary variables. They form a tree that defines all the variables. Every scope has an outer scope. GlobalScope has an outer scope a NullScope. When the lookup reaches a NullScope it answers that the variable is not defined in the code scope. NullScope are reused by simple and clean block

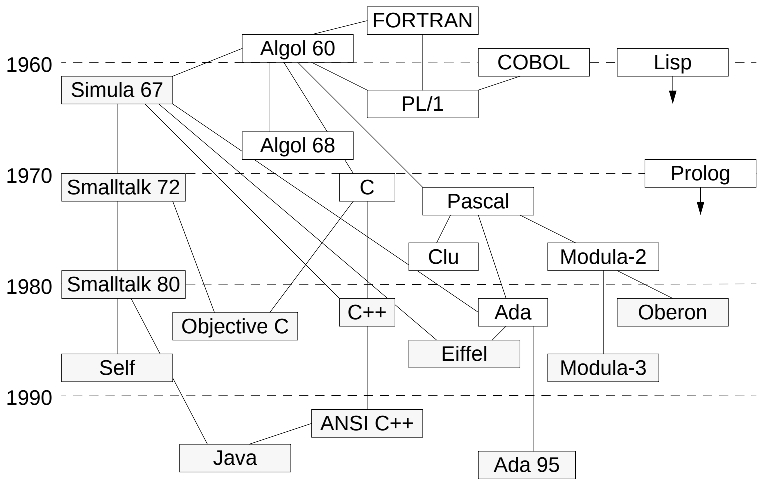
Part IV - Comparisons

- Comparing C++, Java and Smalltalk
- Smalltalk for the Java Programmer
- Smalltalk for the Ada Programmer

24. Comparing C++, Java and Smalltalk

- History
  - target application domains
  - evolution
  - design goals
- Language features
  - syntax
  - semantics
  - implementation technology
- Pragmatics
  - portability
  - interoperability
  - environments & tools
  - development styles

History



Target Application Domains

- Smalltalk
  - Originally conceived as programming language for children.
  - Designed as language and environment for the “Dynabook”.
  - Now: Rapid prototyping. Simulation. Graphical user interfaces. “Elastic” applications.
- C++
  - Originally designed for simulation (C with Simula extensions).
  - Now: Systems programming. Telecommunications and other high-performance domains.
- Java
  - Originally designed for embedded systems.
  - Now: Internet programming. Graphical user interfaces.

Evolution

- Smalltalk
  - Originally (1972) every object was an independent entity. The language evolved to incorporate a meta-reflective architecture.
  - Now the language (Smalltalk-80) is stable, but the environments and frameworks continue to evolve.
- C++
  - Originally called C with classes, inheritance and virtual functions (Simula-like).
  - Since 1985 added strong typing, **`new`** and **`delete`**, multiple inheritance, templates, exceptions, and many other features.
  - Standard libraries and interfaces are emerging. Still evolving.
- Java
  - Originally called Oak, Java 1.0 was already a stable language.
  - Java 1.1 and 1.2 introduced modest language extensions (inner classes being the most important).
  - The Abstract Windowing Toolkit was radically overhauled to support a more general-purpose event model. The libraries are still expanding and evolving.

Language Design Goals

- Smalltalk
  - “Everything is an object”
  - Self-describing environment
  - Tinkerability, a “state of mind”
- C++
  - C with classes with strong-typing
  - “Every C program is also a C++ program” ... almost
  - No hidden costs
- Java
  - C++ minus the complexity (syntactically, not semantically)
  - Simple integration of various OO dimensions (few innovations)
  - “Java — it’s good enough”

Unique, Defining Features

- Smalltalk
  - Meta-reflective architecture
  - The ultimate modelling tool
  - Mature framework technology
- C++
  - “Portable assembler” with HL abstraction mechanisms
  - Programmer is in complete control
  - Templates (computationally complete!)
- Java
  - Dynamically loaded classes, applications are not “installed” in the conventional sense
  - First clean integration of many object-oriented dimensions (concurrency, exceptions ...)

Overview of Features

	Smalltalk	*C++*	*Java*
object model	pure	hybrid	pure
memory management	automatic	manual	automatic
dynamic binding	always	optional	yes (it depends)
inheritance	single	multiple	single
generics	no	templates	no
type checking	dynamic	static	static
modules	namespaces	no (header files)	packages
exceptions	yes	yes (weakly integrated)	yes (well integrated)
concurrency	yes (semaphores)	no (libraries)	yes (monitors)
reflection	fully reflective architecture	limited	limited

Syntax

- Smalltalk
  - Minimal. Essentially there are only objects and messages.
  - A few special operators exist for assignment, statements, blocks, returning etc.
- C++
  - Baroque. 50+ keywords, two commenting styles, 17 precedence levels, opaque type expressions, various syntactic ambiguities.
- Java
  - Simplified C++. Fewer keywords. No operator overloading.

Object Model

- Smalltalk
  - “Everything is an object”
  - Objects are the units of encapsulation
  - Objects are passed by reference
- C++
  - “Everything is a structure”
  - Classes are the units of encapsulation
  - Objects are passed by value
  - Pointers are also values; “references” are really aliases
- Java
  - “Almost everything is an object”
  - Classes are the units of encapsulation (like C++)
  - Objects are passed by reference -> no pointers

Memory Management

- Smalltalk
  - Objects are either primitive, or made of references to other objects
  - No longer referenced objects may be garbage collected, garbage collection can therefore be efficient and non-intrusive
- C++
  - Objects are structures, possibly containing pointers to other objects
  - Destructors should be explicitly programmed (cf. OCF)
  - Automatic objects are automatically destructed
  - Dynamic objects must be explicitly `deleted`
  - Reference counting, garbage collection libraries and tools (Purify) can help
- Java
  - Objects are garbage collected
  - Special care needed for distributed or multi-platform applications!

Dynamic Binding

Smalltalk

- Message sends are always dynamic
  - aggressive optimization performed (automatic inlining, JIT compilation etc.)

C++

- Only virtual methods are dynamically bound
  - explicit inling (but is only a “hint” to the compiler!)
- Overloaded methods are statically disambiguated by the type system
  - Overridden, non-virtuals will be statically bound!
- Overloading, overriding and coercion may interfere!

```
A::f(float); B::f(float), B::f(int); A b = new A; b.f(3) calls A::f(float)
```

Java

- All methods (except “static,” and “final”) are dynamically bound
- Overloading, overriding and coercion can still interfere!

Inheritance, Generics

Smalltalk

- Single inheritance; single root: Object,
- Dynamic typing, therefore no type parameters needed for generic classes

C++

- Multiple inheritance; multi-rooted
- Generics supported by templates (glorified macros)
  - multiple instantiations may lead to “code bloat”

Java

- Single inheritance; single root Object
  - Multiple subtyping (a class can implement multiple interfaces)
- No support for generics; you must explicitly “downcast” (dynamic typecheck)
  - Several experimental extensions implemented ...

Types, Modules

Smalltalk

- Dynamic type-checking
  - invalid sends raise exceptions
- No module concept — classes may be organized into *categories*
  - some implementations support *namespaces*

C++

- Static type-checking
- No module concept
  - use header files to control visibility of names

Java

- Static *and* dynamic type-checking (safe downcasting)
- Classes live inside packages

Exceptions, Concurrency

Smalltalk

- Can signal/catch exceptions
- Multi-threading by instantiating Process
  - synchronization via Semaphores

C++

- Try/catch clauses
  - any value may be thrown
- No concurrency concept (various libraries exist)
  - exceptions are not necessarily caught in the right context!

Java

- Try/catch clauses
  - exception classes are subclasses of Exception or Error
- Multi-threading by instantiating Thread (or a subclass)
  - synchronization by monitors (synchronized classes/methods + wait/signal)
  - exceptions are caught within the thread in which they are raised

Reflection

Smalltalk

- Meta-reflective architecture:
  - every class is a subclass of Object (including Class)
  - every class is an instance of Class (including Object)
  - classes can be created, inspected and modified at run-time
  - Smalltalk's object model itself can be modified

C++

- Run-time reflection only possible with specialized packages
- Compile-time reflection possible with templates

Java

- Standard package supports limited run-time "reflection"
  - only supports *introspection*

Implementation Technology

Smalltalk

Virtual machine running “Smalltalk image.” Classes are compiled to “byte code”, which is then “interpreted” by the VM — now commonly compiled “just-in-time” to native code.

Most of the Java VM techniques were pioneered in Smalltalk.

C++

Originally translated to C. Now native compilers.

Traditional compile and link phases. Can link foreign libraries (if link-compatible.) Opportunities for optimization are limited due to low-level language model.

Templates enable compile-time reflection techniques (i.e., to resolve polymorphism at compile-time; to select optimal versions of algorithms etc.)

Java

Hybrid approach.

Each class is compiled to byte-code. Class files may be dynamically loaded into a Java virtual machine that either interprets the byte-code, or compiles it “just in time” to the target machine.

Standard libraries are statically linked to the Java machine; others must be loaded dynamically.

Portability, Interoperability

Smalltalk

- Portability through virtual machine
- Interoperability through special bytecodes,native methods and middleware

C++

- Portability through language standardization (C as a “portable assembler”)
- Interoperability through C interfaces and middleware

Java

- Portability through virtual machine
- Interoperability through native methods and middleware

Environments and Tools

Advanced development environments exist for all three languages, with class and hierarchy browsers, graphical debuggers, profilers, “make” facilities, version control, configuration management etc.

In addition:

Smalltalk

- Incremental compilation and execution is possible

C++

- Special tools exist to detect memory leaks (e.g., Purify)

Java

- Tools exist to debug multi-threaded applications.

Development Styles

Smalltalk

- Tinkering, growing, rapid prototyping.
- Incremental programming, compilation and debugging.
- Framework-based (vs. standalone applications).

C++

- Conventional programming, compilation and debugging cycles.
- Library-based (rich systems libraries).

Java

- Conventional, but with more standard libraries & frameworks.

The Bottom Line ...

You can implement an OO design in any of the three.

Smalltalk

- Good for rapid development; evolving applications; wrapping
- Requires investment in learning framework technology
- Not suitable for connection to evolving interfaces (need special tools)

C++

- Good for systems programming; control over low-level implementation
- Requires rigid discipline and investment in learning language complexity
- Not suitable for rapid prototyping (too complex)

Java

- Good for internet programming
- Requires investment in learning libraries, toolkits and idioms
- Not suitable for reflective programming (too static)

25. Smalltalk for the Java Programmer

- Syntax
- A bit of semantics

Syntax

- Reference to nowhere

```
null                              nil
```

- Comment

```
/* comment */, // comment        "comment"
```

- Assignment

```
a = 1                             a := 1
```

- Basic types

```
"string"                          'string'
'c'                               $c
true, false                       true, false
```

- Identity and Equality

```
"lulu" =="lulu"                   'lulu' == 'lulu'
"lulu".equals ("lulu")
```

- Self reference

```
this, super                       self, super
this.getClass()                   self class
```


Syntax (II)

- Instance Variables Access

```
x                           x
this.x                      x
anotherObject.x
```

- Instance Variable Definition

```
Node aNode;                 aNode
```

- Local Variable

```
Node aNode;                 | aNode |
```

Syntax - Methods, Conditionals, Loops

- Message Sends

```
anObject.foo()                               anObject foo
this.foo() ; foo()                           self foo
anObject.foo(a,b)                            anObject foo: a with: b
addAll(index, col)                           at: index addAll: col
anObject fooA() ; anObject fooB()            anObject fooA ; fooB
anObject fooA(); anotherObject fooB()      anObject fooA. anotherObject fooB
```

- Method Definition

```
public boolean addAll (int index, Collection aCollection)
                                             at: index addAll: aCollection
```

- Conditionals

```
if (col.isEmpty()) {a}                       col isEmpty ifTrue: [a]
if (col.isEmpty()) {a} else {b}            col isEmpty ifTrue: [a] ifFalse: [b]
while (col.isEmpty()){a}                     [col isEmpty] whileTrue: [a]
do{a} while(col.isEmpty())
```

- Loops

```
for (int n=1; n < k; n++){...n..}            1 to: k do: [:n| ...]
                                             k timesRepeat: [ ]
                                             collection do:, collect:, detect:,
try {a} catch (Exception e) {b}              [a] on: Exception do: [b]
```


No Primitive Types, Only Objects

```
"string"                                'string'
new String ("string")

true                                    true
new Boolean (true)

1
new Integer (1)                         1


int i,j ;                               i + j
i + j

Integer i, j;
i.add(j)
```


Literals representing the same object

```
"a" == "b"                                        'a'=='b'
"a".equals("b")                                   'a' = 'b'
a = "string";                                     a := 'string'.
b = "string";                                     b := 'string'.
c = new String ("string");                        c := #string.
                                                  d := #string
a == b true                                       a = b true
a == c false                                      a == b false
a.equals(c) true                                  c = d true
a.equals(b) true                                  c == d true
```

26. Smalltalk For the Ada Programmer

- Vocabulary
  - package + type -> class
  - subprograms -> methods
  - record component -> instance variable
  - package variable -> classVariable
- Class Definition
- Method Definition
- Instance Creation Method
- Instance Creation

Class Definition

```
with Ada.Strings.Unbounded;    use Ada.Strings.Unbounded;
with Nodes;                    use Nodes;
package Packets is
   type Packet is new Object with private; -- extending the data structure
   ...
private
   type Packet is new Object with record -- the record component
      Contents: Unbounded_String;
      Addressee: Integer;
      Originator: Node;
   end record;
end Packets;

Object subclass: #Packet
   instanceVariableNames: 'contents addressee originator '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LAN-Simulation'
```


Method Definition

```
package Packets is
   type Packet is new Object with private; -- extending the data structure
   function Addressee(A_Packet: Packet) return Integer;
   procedure Addressee (A_Packet: in out Packet, An_Address: in Integer);
   function Is_Sent_By (A_Packet: Packet, A_Node: Node) return Boolean;
   function Is_Addressed_To (A_Packet: Packet, A_Node: Node) return Boolean;
private
   ...
end Packets;

Packet>>addressee
         ^ addressee
Packet>>addressee: aSymbol
         addressee := aSymbol
Packet>>isAddressedTo: aNode
      "returns true if I'm addressed to the node aNode"
      ^ self addressee = aNode name
Packet>>isSentBy: aNode
         ^ originator = aNode
```

Method Definition (II)

```
package body Packets is
    function Addressee (A_Packet: Packet) return Integer is
    begin
        return A_Packet.Addressee;
    end Addressee;
    procedure Addressee (A_Packet: in out Packet, An_Address: in Integer) is
    begin
        A_Packet.Addressee := An_Address;
    end Addressee;
    ...
end Packets;

Packet>>addressee
            ^ addressee
Packet>>addressee: aSymbol
            addressee := aSymbol
```


Method Definition(III)

```
package body Packets is
    ...
    function Is_Sent_By (A_Packet: Packet, A_Node: Node) return Boolean is
    begin
        A_Packet.Originator = A_Node;
    end Is_Sent_By;
    function Is_Addressed_To (A_Packet: Packet, A_Node: Node) return Boolean is
    begin
        A_Packet.Addressee = Name(A_Node); --Name is a function on type Node
    end Is_Addressed_To;
end Packets;

Packet>>isAddressedTo: aNode
        "returns true if I'm addressed to the node aNode"
        ^ self addressee = aNode name

Packet>>isSentBy: aNode
            ^ originator = aNode
```


Instance Creation Method

```
package Packets is
    type Packet is new Object with private; -- extending the data structure
    function Send_To (Contents: String, Address: Integer) return Packet;
    ...
end Packets;
package body Packets is
    ...
    function Send_To (Contents: String, Address: Integer) return Packet;
    begin
    return (To_Unbounded(Contents), Integer, Empty_Node);
    end Send_To;
end Packets;

Packet class>>send: aString to: anAddress
    |inst|
    inst := self new.
    inst contents: aString.
    inst to: anAddress.
    ^inst
```

Instance Creation

```
procedure XXX
   P: Packet := Send_To (“This packet travelled to the printer“, 123);
begin
   Addressee(P);
   ...
end XXX;


XXX
   |p|
   p := Packet send:  'This packet travelled to the printer' to: 123.
   p addressee
```

27. References

A Jungle of Names

Some Smalltalk Dialects:

- Smalltalk-80 -> ObjectWorks -> VisualWorks by (ParcPlace -> ObjectShare->Cincom)
  mac, pc, hp, linux, unix
  www.cincom.com/visualworks/
- IBM Smalltalk (pc, unix, aix...)
  www.software.ibm.com/ad/smalltalk/
- Smalltalk-V (virtual) -> Parts -> VisualSmalltalk by (Digitalk -> ObjectShare)
- VisualAge = IBMSmalltalk + Envy (OTI -> IBM)
- Smalltalk Agents (Mac) www.quasar.com
- SmallScript www.quasar.com (.Net, PC and Mac)
- Smalltalk MT (PC, assembler)
- Dolphin Smalltalk (PC)
  www.object-arts.com/Home.htm
- Smalltalk/X -> www.exept.de (run java byte code into Smalltalk VM)
- Smalltalk/Express (free now but not maintained anymore)
- Enfin Smalltalk -> Object Studio (Cincom)
  www.cincom.com/objectstudio/

Team Development Environments

- Envy (OTI) most popular, available for VisualWorks
- VSE (Digitalk), (not available)
- TeamV, (not available)
- Store (new Objectshare)
- ObjectStudio v6 (similar to Envy)

Some Free Smalltalks

Professional Environment

- VisualWorks 3.0 and VW5i.2 on PC for free
- VisualWorks 3.0 and VW5i.2 on Linux (Red-Hat)
  www.cincom.com
- Dolphin Smalltalk on PC (not the last version)
  www.object-arts.com/Home.htm

New concepts

- Squeak (Morphic Objects + Socket + all Platforms) continous development
  http://www.squeak.org/
- Gnu Smalltalk (not evaluated)

Free for Universities:

- VisualWorks 3.0 and VW5i.2) all platforms and products (www.cincom.com/vwnc/)
- VisualAge is free for University:
  www.software.ibm.com/ad/smalltalk/education/univagr.html
- Envy is free for University
  contact amy_divis@oti.com

Main References

- *Smalltalk: an Introduction to application development using VisualWorks*, T. Hopkins and B. Horan, Prentice-Hall,1995, 0-13-318387-4
- *Smalltalk, programmation orientée objet et développement d'applications*, X. Briffault and G. Sabah, Eyrolles, Paris. 2-212-08914-7
- *On To Smalltalk*, P. Winston, Addison-Wesley, 1998, 0-201-49827-8
- *Smalltalk by Example : The Developer's Guide*, A. Sharp, McGraw Hill, ISBN: 0079130364, 1997
- *Smalltalk Best Practice Patterns*, K. Beck, Prentice Hall, 1997, ISBN 0-13-476904-x
- *Smalltalk with Style*, S. Skublics and E. Klimas and D. Thomas, Prentice-Hall, 1996, 0-13-165549-3.
- *The Smalltalk Developer's Guide to VisualWorks*, T. Howard, Sigs Books, 1995, 1-884842-11-9
- *Mastering Envy/Developer*, J. Pelrine, A. Knight and A. Chou..., SIG Press.
- *The Design Patterns Smalltalk Companion*, S. Alpert and K. Brown and B. Woolf, Addison-Wesley, 1998,0-201-18462-1

Other References

** Smalltalk-80: The language, Adele Goldberg and David Robson, Addison-Wesley, 1984-1989, 0-201-13688-0 (Purple book ST-80, part of the original blue book). VW. old but still really interesting: a reference!

• An introduction to Object-Oriented Programming and Smalltalk, Lewis J. Pinson and Richard S. Wiener, 1988, Addison-Wesley, ISBN 0-201-119127. (ST-80)

• Object-Oriented Programming with C++ and Smalltalk, Caleb Drake, Prentice Hall, 1998, 0-13-103797-8

+ Smalltalk, Objects and Design, Chamond Liu, Manning, 0-13-268335-0 (IBM Smalltalk)

+ Smalltalk the Language, David Smith, Benjamin/Cummings Publishing, 1995,0-8053-0908-X (IBM smalltalk)

• Discovering Smalltalk, John Pugh, 94 (Digitalk Smalltalk)

• Inside Smalltalk (I & II), Wilf Lalonde and Pugh, Prentice Hall,90, (ParcPlace ST-80)

• Smalltalk-80: Bits of History and Words of Advice, G. Kranser, Addison-Wesley,89, 0-201-11669-3

Other References (II)

- The Taste of Smalltalk, Ted Kaehler and Dave Patterson, Norton, 0-393-95505-2,1985
- Smalltalk The Language and Its Implementation (contains the original VM description available at users.ipa.net/~dwighth/smalltalk/bluebook/), Adele Goldberg and Dave Robson, 0-201-11371-6, 1982 (called The Blue Book)

To understand the language, its design, its intention....

- Peter Deutsch, The Past, The Present and the Future of Smalltalk, ECOOP'89
- Byte 81 Special Issues on Smalltalk (read Dan Ingalls paper on language intent)
- Alan Kay, The Early History of Smalltalk, History of Programming Languages, Addison-Wesley, 1996