Smalltalk — a Pure Object Language and its Environment

Dr. S. Ducasse

University of Bern
Winter semester (W7070)
1998/99

1. Smalltalk Concepts

Smalltalk: More than a Language

- A small and uniform language (two days for learning the syntax).
- A set of reusable classes (basic data structure, UI, database accesses...).
- A set of powerfull development tools (Browsers, UIBuilder, Inspector, changes, crash recovery, projects).
- A run-time environment based on Virtual Machine technology.
- With Envy team working + application management (release, versioning, deployement).

A Jungle of Names

Some Smalltalk Dialects:

- Smalltalk-80 -> ObjectWorks -> VisualWorks by (ParcPlace -> ObjectShare)
- IBM Smalltalk
- Smalltalk-V (virtual) -> Parts -> VisualSmalltalk by (Digitalk -> ObjectShare)
- VisualAge -> IBMSmalltalk + Envy (OTI -> IBM)
- Smalltalk Agents (Mac)
- Smalltalk MT (PC)
- Dolphin Smalltalk (PC)
- Smalltalk/X, Enfin Smalltalk (Cimcon)

Team Development Environment:

- Envy (OTI), VSE (Digitalk), TeamV

Free Software:

- Gnu Smalltalk (no UI), Little Smalltalk (no UI): Do not use them!!
- -> Squeak (Morphic Objects + Socket + all Platforms) under development but amazing
- -> VisualWorks 3.0 on PC for free
- -> VisualWorks 3.0 on Linux (Red-Hat)

Inspiration

"making simple things Very simple and complex things Very possible" [Kay]

- Flex (Alan Kay 1969)
- Lisp (interpreter, blocks, garbage collector)
- Turtle graphics (Logo Project, children programming)
- Direct manipulation interfaces (Sketchpad 1960)
- Simula (classes and message sending, description of a real phenomen by means of a specification language: modeling)

-> DynaBook: a desktop computer for children

Precursor, Innovative and Visionary

- First graphical bitmap-based
  multi-windowing (overlapping windows)
  programming environment (debugger, compiler, editor, browser)
  with a pointing device
  Yes a mouse !!!!

Xerox Smalltalk Team developed the mouse technology and the bitmap:
it was revolutionary! MacIntosh copied them.

- Virtual Machine +
  Plateform independent image technology

- Garbage Collector

- Just in Time Compilation

History

Internal.

1972: First interpreter, more agents than objects (every objects can specify its own syntax).

1976: Redesign: Hierarchy of classes with unique root + fixed syntax (compact byte code), contexts, process + semaphores + Browser + UI class library.

Projects: ThingLab, Visual Programming Environment Programming by Rehearsal.

1978: Experimentation with 8086 microprocessor (NoteTaker project).

External.

1980: Smalltalk-80 (Ascii, cleaning primitives for portability, metaclasses, blocks as first-class objects, MVC,)

Projects: Galley Editor (mixing text, painting and animations) + Alternate Reality Kit (physics simulation)

1981: books + four external virtual machines (Dec, Apple, HP and Tektronix) -> gc by generation scavenging

1988: Creation of Parc Place Systems

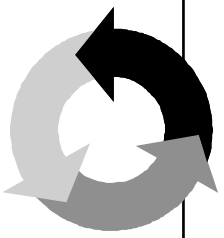
1992: Draft Ansi

1995-6: New Smalltalk implementations (MT, dolphin, Squeak VM in Smalltalk....)

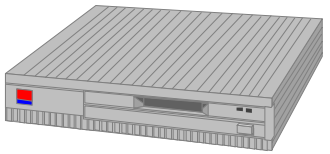
Source, Virtual Machine, Image and Changes

All the objects of the system
at a moment in time

IMAGE1.IM
IMAGE1.CHA



A byte-code interpreter:
the virtual machine interpretes the image



+

Standard SOURCES

Shared by everybody

IMAGE2.IM
IMAGE2.CHA

One per user

Smalltalk's Concepts

- Everything is an object (numbers, files, editors, compilers, points, tools, boolean).
- Objects only communicate by message passing.
- Each object is an instance of one class (that is an object too).
- A class defines the structure and the behavior of its instances.
- Each object possesses its own set of values.

Programming in Smalltalk: Reading an Open 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

Messages, Methods and Protocols

Message: **What** behavior to perform

```
aWindow openAroundCursorWithExtent: 0@0 extent: 100@100 andType: #normal
```

Method: **How** to carry out the behavior (.....)

```
openAroundCursorWithExtent: extent andType: aType
| pt box |
pt := WindowSensor cursorPoint.
box := pt - (extent // 2) extent: extent.
self openIn: box withType: aType
```

Protocol: The complete set of messages an object responds to

```
#close, #damageRepairIsLazy:, #finishOpening,
#noticeOfWindowClose, #release, #beMaster, #bePartner, #beSlave,
#checkForEvents, #receiveWindowEvents, #receiveWindowEvents:
#sendWindowEvents #sendWindowEvents: #windowEventBlock
#windowEventBlock:, #openAroundCursorWithExtent:andType:
#openIn:withType:, #openWithExtent:andType:, #openWithType:,
#dropTargetForSource:, #findObjectInterestedInDropAt:forSource:
```

Objects, Classes and Metaclasses

- Every object is instance of a class
- A class specifies the structure and the behavior of all its instances
- Instances of a class share the same behavior and have specific state

- Classes are objects that create other instances
- Metaclasses are just classes that create classes as instances
- Metaclasses described then class behavior and state (subclasses, method dictionary, instance variables...)

Main References

+ (Intro) Smalltalk: an Introduction to application development using VisualWorks, Trevor Hopkins and Bernard Horan, Prentice-Hall,1995, 0-13-318387-4

+ (Intro) Smalltalk, programmation orientée objet et développement d'applications, X. Briffault and G. Sabah, Eyrolles, Paris. 2-212-08914-7

+ (Intro) On To Smalltalk, P. Winston, Addison-Wesley, 1998, 0-201-49827-8

+ (Idioms) Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997, isbn 0-13-476904-x

Praxisnahe Gebrauchsmuster, K. Beck, Prentice-Hall, 1997, ISBN 3-8272-9549-1

+ (Idioms) Smalltalk with Style, S. Skublics and E. Klimas and D. Thomas, Prentice-Hall, 1996, 0-13-165549-3.

+(User Interface Reference) The Smalltalk Developer's Guide to VisualWorks, Tim Howard, Sigs Books, 1995, 1-884842-11-9

+(Design) The Design Patterns Smalltalk Companion, S. Alpert and K. Brown and B. Woolf, Addison-Wesley, 1998,0-201-18462-1

Other References (Old or Other Dialects)

Before buying a book ask me or consult the annotated bibliography on the lecture web page. Do not buy a book with translated **code** in German!

+ (old but a reference) 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)

• 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-Prentice-Hall, 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, 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
- Alan Kay, The Early History of Smalltalk, History of Porgamming Languages, Addsison-Wesley, 1996

2. The Taste of Smalltalk

Two examples:

- hello world
- a small LAN simulator

To give you an idea of:

- the syntax
- the elementary objects and classes
- the enviromnent

Some Conventions and Precisions

- Code `Transcript show: 'Hello world'`
- Return Value

  `1 + 3` *-> 4*

  `Node new` *-> aNode*

  `Node new` *PrIt-> a Workstation with name:#pc and next node:#mac*
- Method selector `#add:`
- Method scope

  `Node>>``accept: aPacket`

  instance method defined in the class `Node`

  `Node class>>``withName: aSymbol`

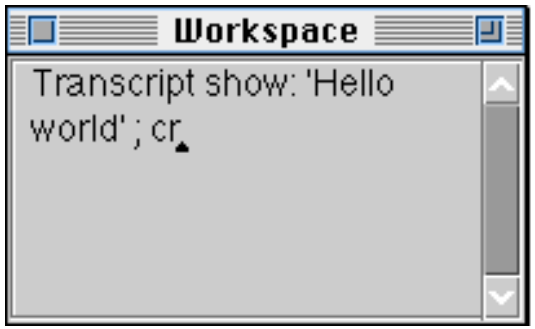
  class method defined in the class `Node` (in the class of the class Node)
- `aSomething` is an instance of the class `Something`
- DoIt, PrintIt and Accept

  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:`)

Hello World!




```
Transcript show: ‘hello world’
```

During implementation, we can dynamically ask the interpreter to evaluate expression. To evaluate an expression, select it and with the middle mous button apply **doIt**.

Everything is an object

The launcher is an object.
The icons are objects.
The workspace is an object.
The window is an object: instance of ApplicationWindow.
The text editor is an object: instance of ParagraphEditor.
The scrollbars are objects too.
'hello word' is an object: aString instance of String.
#show: is a Symbol that is also an object.
The mouse is an object.
The parser is an object instance of Parser.
The compiler is also an object instance of Compiler.
The process scheduler is also an object.
The garbage collector is an object: instance of MemoryObject.
...
=> a world **consistent**, **uniform** written in itself!
you can **learn** how it is implemented, you can **extend** it or even **modify** it.
=> Book concept.

Objects communicate via messages

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

The above expression is a message:

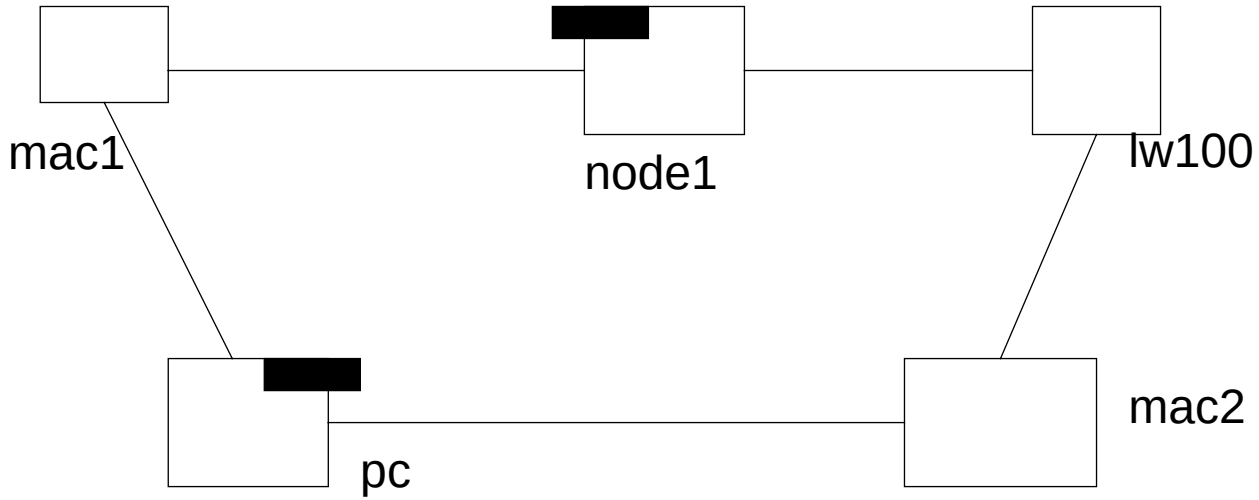
- the object `Transcript` is the receiver of the message
- the selector of the message is `#show:`
- an argument: a string `'hello world'`

```
Transcript
```

is a global variable (starts with an uppercase letter)
that refers to the Launcher's report part.

A LAN Simulator

A LAN contains nodes, workstations, printers, file servers.
Packets are sent in a LAN and the nodes treat them differently.



Once the Classes Defined

We can invoke the following expressions in a workspace or define it as a method.

```
|mac1 pc node1 printer mac2 packet|
"nodes definition"
mac1 := Workstation withName: #Mac1.
pc := Workstation withName: #pc.
node1 := Node withName: #node1.
printer := Printer withName: #lw100.
mac2 := Workstation withName: #Mac2.

"Node connections"
mac nextNode: node1.
node1 nextNode: printer.
printer nextNode: mac2.
mac2 nextNode: pc.
pc nextNode: mac1.
"create a packet and start simulation"
packet := Packet send:  'This packet travelled around to the printer' to:  #lw100.
mac2 originate: packet.
```

(temporary, comments, classes, instance creation class methods, symbol, instance methods, string, sequence, classes start with uppercase letter, temporary not)

A Step Further: Two Printer Methods

```
PrinterServer>>print: thePacket
   "print the packet. Write this on the transcript"
   Transcript show: 'printer ',
                       self name printString,
                       ' printing the packet with contents: ',
                       thePacket contents printString ; cr
```

In C++, Java: we would write

```
      void Printer::print(thePacket Packet)
            ....
PrinterServer>>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]
```

Printer>> is an home made convention to precise the scope of the method (method definition, invoking a method one myself, invoking an overriden method, conditionals, message composition, blocks, parentheses)

3. Syntax and Messages

Everything new meets with resistance
Russian Proverb

The syntax of Smalltalk is really simple and uniform

- Literals: numbers, strings, arrays....
- Variables names
- Pseudo-variables
- Assignment, return
- Message Expressions
- Block expressions

Literals

Numbers: SmallInteger, Fraction, Float, Double

```
1232, 3/4, 4, 2.4e7, 2r101
```

Characters:

```
$F, Character space, Character tab, Character cr
```

Strings:

```
'This packet travelled around to the printer' 'l''idiot'
To introduce a single quote inside a string just double it.
```

Symbols:

```
#class #Mac1 #at:put:  #+
```

Arrays:

```
#(1 2 3) #('lulu' (1 2 3))  #('lulu' #(1 2 3)) #(lulu toto titi)
```

The last array is an array of symbols. When one prints it it shows #(#lulu #toto #titi)

Byte Array:

```
#[1 2 255]
```

Comments:

```
"This is a comment"
```

A comment can be more several lines. Moreover, avoid to put a space between the “ and the first letter. Indeed when there is no space, the system helps you to select a commented expression. You just go after the “ character and double click: all the commented expression is selected. After you can printIt or doIt.

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')
```

Literal or not

```
#(...) considers element as literals
#( 1 + 2 ) PrIt-> #(1 #+ 2)
Array with: (1 +2) PrIt->  #(3)
```

About Literals for the CuriousNote about literals. Literature (Goldberg book) defines a literal as an object which value refer always to the same objet. This is a first approximation to present the concept. However, if we check the literals according to this principle, this is false in VisualWorks (VisualAge as a safer

definition.) Literature defines literals as numbers, characters, strings of character, 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 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 a part of a compiled method that stores the literals used by the methods. You can inspect a class->methodDict-> aCompiledMethod to see.

The following example can illustrate the difference between the literal array and a newly created instance of Array created via Array new. Let us defined 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.

```
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 literals at compile-time and not stored into the literal frame of the compiled method. You can find yourself this infomation by defining these methods on a class, inspecting the class then its method dictionary and then the corresponding methods.

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 not
- Only created using #symbol

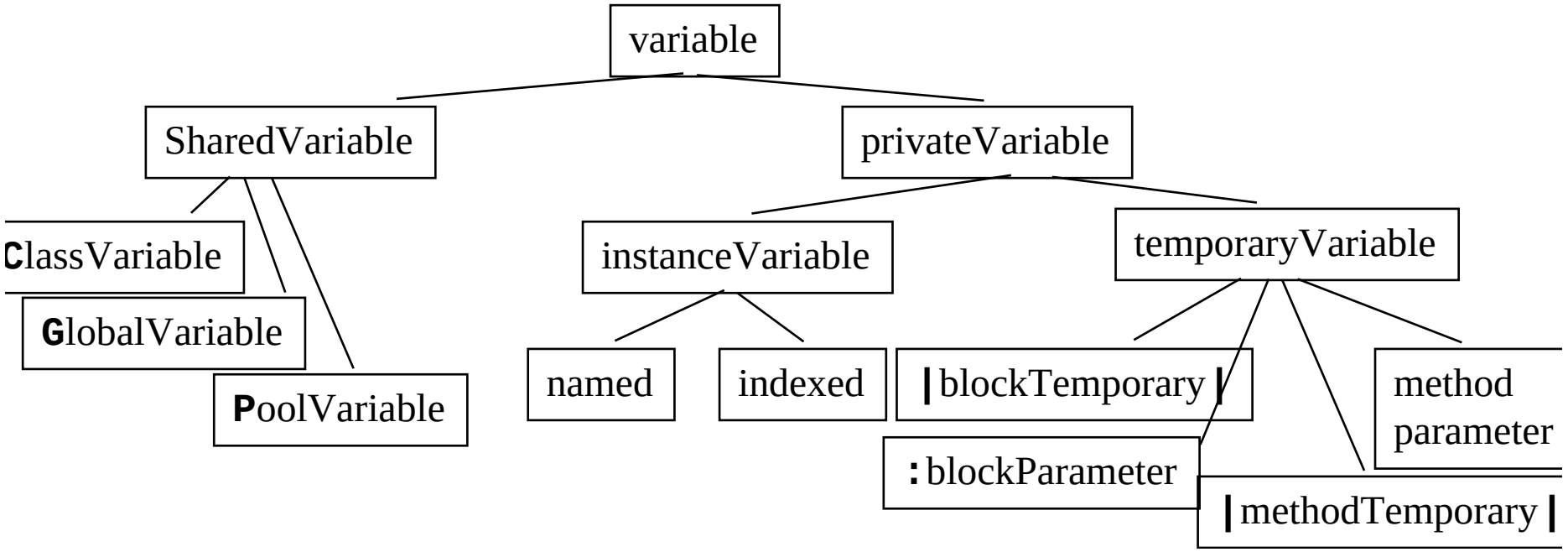
```
#lulu == #lulu  PrIt-> true
'lulu' == 'lulu' PrIt-> false

#lulu, #zeBest PrIt-> 'luluzeBest'
```

Comparing strings is a factor of 5 to 10 slower than symbols. But converting a string to a symbol is more than 100 times more expensive.

Symbols are good candidates for identity based dictionary (`IdentityDictionary`)

Variables

- Maintain a reference to an object
- Untyped, can reference different types of objects
- Shared or private



Temporary Variables

To hold temporary values during evaluation (method execution)

Can be accessed by the expressions composing the method

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

- Avoid to use the same name for a temporary variable than an argument, an instance variable or another temporary variable or block temporary. Your code will be more portable.

Instead of

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

Write

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

- Avoid to use the same temporary variable for referencing two different objects.

Assigments

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

But assignment is not done by message passing.
This is one of the few Smalltalk syntax element

```
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

```
add: newObject after: oldObject
"Add the argument newObject as an element of the receiver in the position just succeeding oldObject"
```

In C++ or Java:

```
“Object” Printer::addafter(newObject “Object”, oldObject “Object”)
```

- But their values cannot be reassigned within the method.

Invalid Example, assuming name is an instance variable:

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

Valid Example

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

Instance Variables

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

Declaration

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

Scope

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

But preferably accessed with accessors

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

Six pseudo-variables (i)

Smalltalk expressions make references to these variables, but we cannot change their value. They are hardwired in the compiler.

- `nil` (nothing) value for the uninitialized variables. Unique instance of the class `UndefinedObject`
- `true` unique instance of the class `True`
- `false` unique instance of the class `False`

Take care

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

Produces an error, but

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

works

(see most common bugs)

Six pseudo-variables (ii)

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

- `self` in the method body refers to the **receiver** of a message.

- `super` in the method body refers also to the **receiver** of the message but its semantics affects the lookup of the method. It starts in the superclass of the class in which the method where the `super` was used and NOT the superclass of the receiver (see method lookup semantics)

```
PrinterServer>>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]
```

- `thisContext` refers to the instance of `MethodContext` that represents the context of a method (receiver, sender, method, pc, stack). Exists only in VisualWorks.

Global Variables

- Capitalized

```
MyGlobal := 3.14
```

Smalltalk will prompt you.

```
Smalltalk at: #MyGlobal put: 3.14
Global PrIt-> 3.14
Smalltalk at: #MyGlobal PrIt-> 3.14
```

- Store in the default environment: `Smalltalk` (a`SystemDictionary`)
- Accessible from everywhere
- Usually not really a good idea to use them, use a classVariable (if shared within an hierarchy or a instance variable of a class)
- To remove a global variable:

```
Smalltalk removeKey: #MyGlobal
```

- Some predefined global variables:

```
Smalltalk (classes + globals)
Undeclared (a Pool dictionary 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

```
2.4 inspect
```

Binary

```
1 + 2 -> 3
```

Keyword based

```
6 gcd: 24 PrIt-> 6
```

```
(1 + 2) * (2 + 3) PrIt-> 15
```

Message composed by :

- a receiver always evaluated (1+2)
- a selector never evaluated
- and a list possibly empty of arguments that are all evaluated (2+3)

Receiver linked with `self` in a method body

Message = Effect + Return

Three kind of message actions:

Affects the receiver and returns

```
pc nextNode: mac1 PrIt-> aWorkstation
```

```
Date new day: 12 year: 1997
   PrIt-> January 12, 1997
```

Only returns a new object

```
3.14 truncated PrIt-> 3
Workstation withName: #Mac1 PrIt-> aWorkstation
```

Perform side effect and returns

```
Browser browseAllSendersOf: #open:label:
   PrIt-> aBrowser + open a Browser
```

Unary Messages

aReceiver aSelector

```
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

Binary messages:

- arithmetic, comparison and logical operations
- one or two characters long taken from

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

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

Restriction:

- second character is never $-
- no mathematical precedence

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

Keyword Messages

receiver keyword1: argument1 keyword2: argument2 ...

In C-like languages: receiver keyword1keyword2...(argument1 type1, argument2, type2) : return-type

```
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']

Workstation withName: #Mac2
mac nextNode: node1.
Packet send:  'This packet travelled around to the printer' to:  #lw100.
```

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)
```

Hints: Put () when two keyword based messages are consequent. Else the precedence order is fine.

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

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

Sequence

message1.
message2.
message3

```
Transcript cr.
Transcript show: 1 printString.
Transcript cr.
Transcript show: 2 printString

|mac1 pc node1 printer mac2 packet|
   "nodes definition"
mac1 := Workstation withName: #Mac1.
pc := Workstation withName: #pc.
node1 := Node withName: #node1
```

Cascade

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

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

Equivalent to:

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

Important: the semantics of the cascade is to send all the messages composing the cascade to the receiver of the FIRST message being involved into the cascade.

In the following example the FIRST message being involved in a cascade is the first `#add:` and not `#with:`. So all the messages will be sent to the result of the parenthesed expression the newly created instance `anOrderedCollection`

```
(OrderedCollection with: 1) add: 25; add: 35
Workstation new name: #mac ; nextNode: aNode
```

`name:` is sent to the newly created instance of workstation and `nextNode:` too.

yourself

One problem:

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

Returns 35 and not the collection!

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)
```

Have You Really Understood Yourself ?

Yourself returns the receiver of the cascade:

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

Here the receiver of the cascade is a`Workstation` the newly created instance and not the class `Workstation`

In

```
(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
```

Block (i): Definition

- A deferred sequence of actions
- Return values is the result of the last expression of the block
- = Lisp Lambda-Expression, ~ C functions

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

Two blocks without variables and temporary

```
PrinterServer>>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]
```

Block (ii): Evaluation

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

Blocks are first class objects, they are created, pass 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
```

Block (iii)

```
|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 reason avoid as much as possible to refer to variables that are outside a block.

Syntax Summary (i)

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]`
point: `10@120`
integer: `1`
real: `1.5, 6.03e-34,4, 2.4e7, 2r101`
float: `1/33`
boolean: `true, false`

Syntax Summary (ii)

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

unary message:	receiver selector
binary message:	receiver selector selector
keyword based:	receiver keyword1: arg1 keyword2: arg2...
cascade:	message ; selector ...
sequence:	message . message
result:	^
parenthesis:	`(...)`
byte array:	`#[...]`

What You Should Know

- Syntax
- Basic objects
- Message constituants
- Message semantics
- Message precedence
- Block definition
- Block use
- yourself semantics
- pseudo-variables

To know all that, the best thing to do is to take a Smalltalk and type in some expressions, to look at the return expressions

4. Basic Objects, Conditional and Loops

- Common Shared Behavior (minimal version)
- Number (subclass of `Magnitude`)

```
Fraction
Integer
    LargeInteger
    SmallInteger
LimitedPrecisionReal
    Double
    Float
```

- `Boolean`: superclass of `True` and `False`

- Collection super of more than 80 classes:
  (`Bag`, `Array`, `OrderedCollection`, `SortedCollection`, `Set`, `Dictionary`…)

- Loops and Iteration abstraction

- Streams and Files

Common Shared Behavior (i)

Object class is the root of inheritance tree

Defines the common and minimal behavior for all the objects in the system.

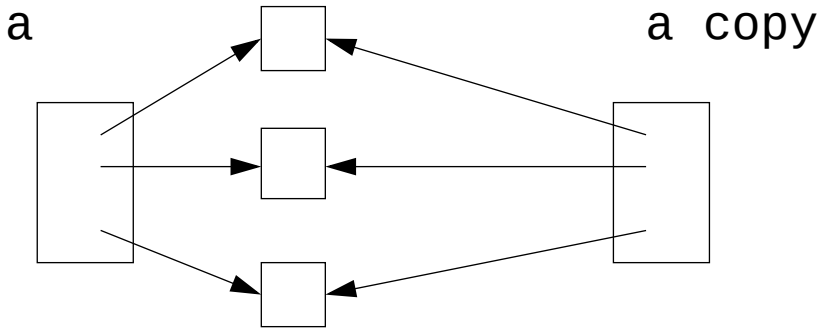
=> 161 instance methods + 19 class methods

- `#class`
- Comparison of objects: `#==, #~~, #=, #=~, #isNil, #notNil`

- Copy 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 overriden. 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!
(See most common bugs)

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)'
```

Create instances from stored objects: class methods `readFrom: aStream, readFromString: aString`

```
Object readFromString:  '((OrderedCollection new) add: 4; add: 3; yourself)'
 -> OrderedCollection (4 3)
```

Notifying the programmer:

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

Examing Objects: `#browse, #inspect`

Essential Common Shared Behavior

`#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 whom evaluation recreates an object equal to the receiver

`#readFromString: aStream` recreates an object

Basics on Number

- Arithemic

```
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 on Numbers: Double Dispatch (i)

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

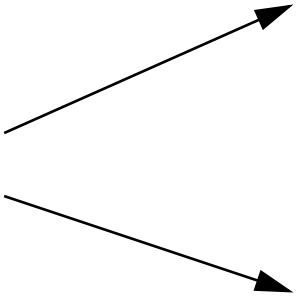
Example: Coercion between Float and Integer

A not 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]
```

Deeper on 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 on 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 numbers)."

    ^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 on 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'
```

Boolean Objects

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

- uniform
- but optimized, inlined

- Logical Comparisons &, |, xor:, not

```
aBooleanExpression comparison anotherBooleanExpression
```

```
(1 isZero) & false
```

- Lazy Logical operators

```
aBooleanExpression and: aBlock
```

aBlock will only be valued if aBooleanExpression is `true`

```
false and: [1 error: 'crazy'] PrIt-> false
```

```
aBooleanExpression or: aBlock
```

aBlock will only be valued if aBooleanExpression is false

Boolean Objects and Conditionals

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

```
1 < 2 ifTrue: [...] ifFalse: [...]
1 < 2 ifFalse: [...] ifTrue: [...]
1 < 2 ifTrue: [...]
1 < 2 ifFalse: [...]
```

Take care `true` is the boolean value and `True` is the class of `true` its unique instance!

Note: Why conditional expressions use blocks?

Because, when a message is sent: the receiver and the arguments of the message are evaluated. So block uses are necessary to avoid to evaluate both branchs.

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

whileTrue:

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

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

timesRepeat:

```
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

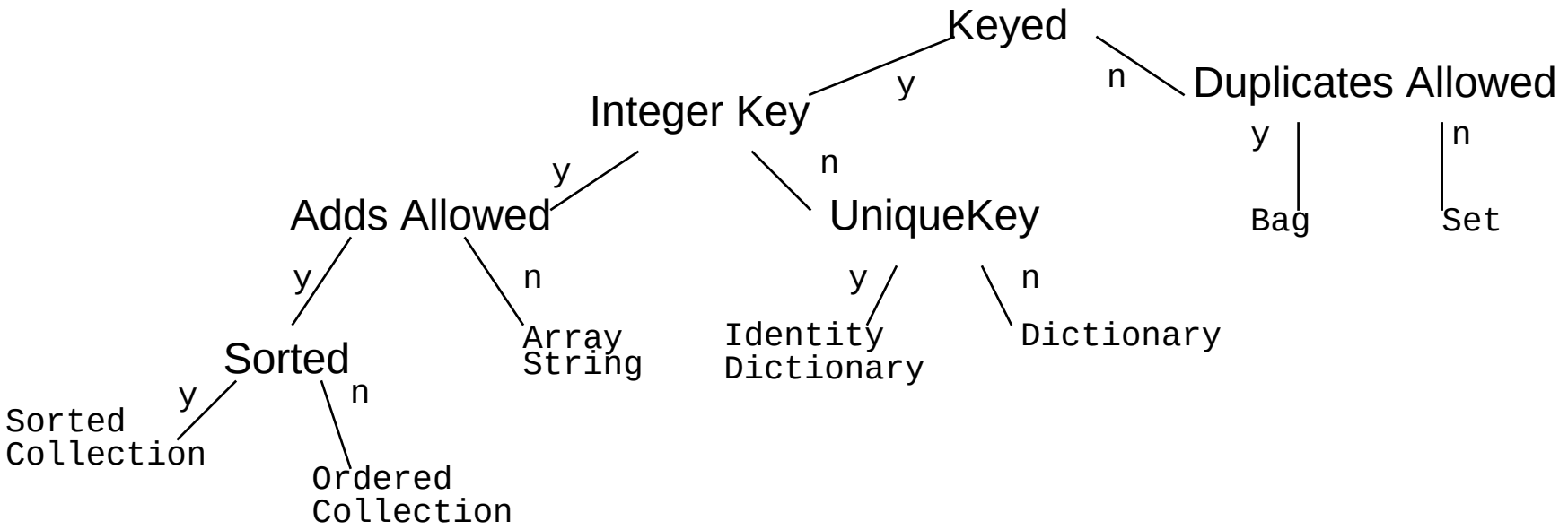
- Only the most important
- Some criterias to identify them. Access: indexed, sequential or key-based.

Size: fixed or dynamic. Element type: any or well-defined type.

Order: defined, defineable or no. Duplicate: 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
```

Another View



Collection Methods

Will be **defined, redefined, optimized or forbiden** in 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)

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`

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

KeyedCollection Specific (Dictionary)

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`

```
|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')
```

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
```

Sure!

Or

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

And contrary to the first solution, this solution works well for indexable collection 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
```

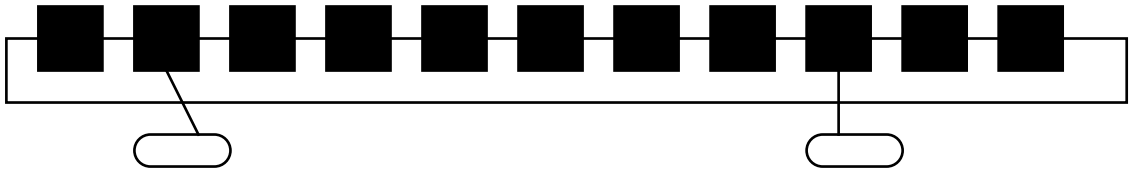
Collection Abstraction

```
aCollection includes: anElement
aCollection size
aCollection isEmpty
```

```
#(1 2 3 4 5) contains: 4 -> true
#(1 2 3 4 5) size -> 5
#(1 2 3 4 5) isEmpty -> false
```

Streams

- Allows the traversal of a collection
- Associated with a collection
  - collection is a Smalltalk collection: `InternalStream`
  - 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
```

printSring, 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(i)

Stream.

`#next` returns the next element
`#next: n` returns the n next elements
`#contents` returns all the elements
`#nextPut: anElement` inserts element at the next position
`#nextPutAll: aCollection` inserts the collection element from the next position
`#atEnd` returns true if at the end of the collection

PeekableStream.

Access to the current without passing to the next
`#peek`
`#skipFor: anAgrument`
`#skip: n` increases the position of n
`#skipUpTo: anElement` increases the position after anElement
Creation
`#on: aCollection,`
`#on: aCol from: firstIndex to: lastIndex` (index elements included)

Stream Classes (ii)

PositionnableStream

`#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 and 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 and Files

Filename.

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

Example: removing Smalltalk comments of 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
```

What you should know

- Number protocol
- Boolean protocol
- Collection protocol
- Loops
- Conditional
- Iteration Abstraction
- Collection Abstraction

But the best way to know that is to play with a Smalltalk interpreter! Yes again!

5. Dealing with Classes

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

Class Definition

A template is proposed by the browser:

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

Example

```
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 / inheritance
- Default value is `nil`

- Private to the instance: instance based (C++ class-based)
- Can be accessed by all the methods of the class and subclasses (instance methods)
- **But instance variables cannot be accessed by class methods.**
- A client cannot directly access to iv. No private, protected like in C++
- Accessing methods to access instance variable.

classVariable

- A pretty bad name: should have been called **Shared Variables**
- Begins with a uppercase letter
- a classVariable is **shared** and directly accessible by all the instances and subclasses
- a classVariable can be directly accessed in instance methods and class methods

```
NameOfSuperclass subclass: #NameOfClass
   ...
   classVariableNames: 'ClassVarName1 ClassVarName2'
   ...

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

Pay attention and be sure than you really need a classVariable before using it!

- Often classVariable can be replaced by class methods

```
Packet class>>domain
     ^ ‘iam.unibe.ch’
```

Class Instance Variables / ClassVariable

- a classVariable is **shared** and directly accessible by all the instances and subclasses

- Class instance variables as 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 all the inheritance tree is impacted!

poolVariables

- Also called Pool Variables.
- 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.

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, never use `poolDictionaries`

Method Definition

A template is proposed by the browser

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

   | temporary variable names |
   statements
```

Example from `PrinterServer`

You type:

```
accept: thePacket
   "If the packet is addressed to me, I print it. Else I just behave like a node”

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

I show

```
PrinterServer>>accept: thePacket
```

Iv Access Example

```
Packet>>isOriginatingFrom: aNode

   ^ self originator = aNode
```

is equivalent to:

```
Packet>>isOriginatingFrom: aNode

   ^ originator = aNode
```

Accessors are interesting to implement lazzy initialization

A lazzy initialization schema:

```
Packet>>originator
   originator isNil
      ifTrue: [originator := Node new]
   ^ originator
```

Return Value

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

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

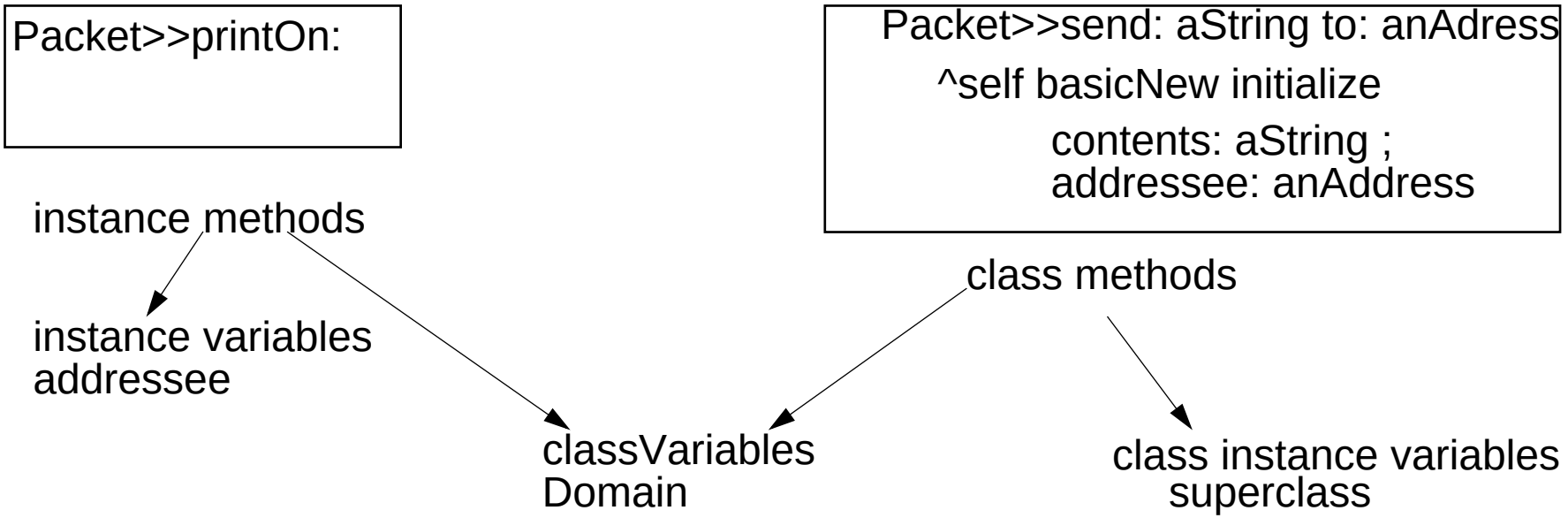
is equivalent to

```
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:`

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

Visibility of Variables



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>>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
```

Quick Naming Conventions

- Shared variables begin with an upper case letter
- Private variables begin with a lower case letter
- Use imperative for methods performing action `#openOn:`

For accessor, use the same name as for the instance variable

```
upperLimit
    ^ upperLimit

upperLimit: aNumber
    upperLimit := aNumber
```

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

```
isNil, hasBorder, isEmpty
```

- 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 of the type of the receiver.

Message Sending & Method Lookup

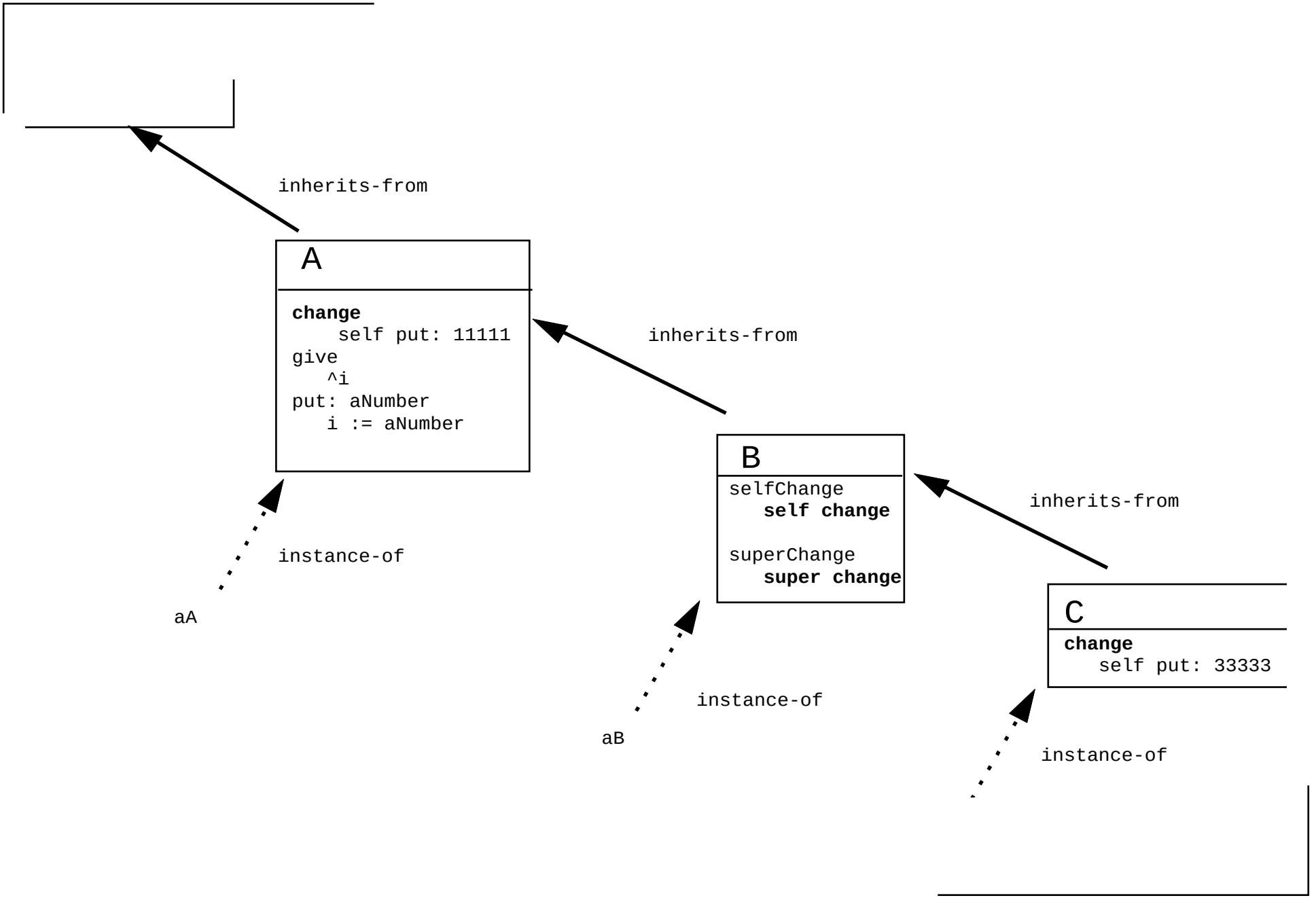
sending a message: receiver selector args <=>
applying a method looked up associated with selector to the receiver and the args

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

=> the lookup starts in the class of the receiver.
If the method is defined in the class dictionary, it is returned.

Else the search continues in the superclasses of the receiver's class.

If no method is found and that 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.


inherits-from
A
change
self put: 11111
give
^i
put: aNumber
i := aNumber
inherits-from
B
selfChange
self change
superChange
super change
inherits-from
C
change
self put: 33333
instance-of
aA
instance-of
aB
instance-of

Example

```
Object subclass: #A
       instanceVariableNames: 'i '...
A>>change
   self put: 11111
A>>give
   ^i
A>>put: aNumber
   i := aNumber

A subclass: #B ...
B>>selfChange
   self change
B>>superChange
   super change

B subclass: #C
C>>change
   self put: 33333
```

Run the messages

```
aA change give -> 11111
aB change give -> 11111
aC change give -> 33333

aC selfchange give -> 33333
aC superchange give -> 11111
aB selfchange give -> 11111
aB superchange give -> 11111
```

Semantics of super

- As `self`, `super` is a pseudo-variable that refers to the receiver of the message.
- Used to invoke overriden 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.

Other said:

- `super` causes the method lookup to begin searching in the superclass of the class of the method containing `super`

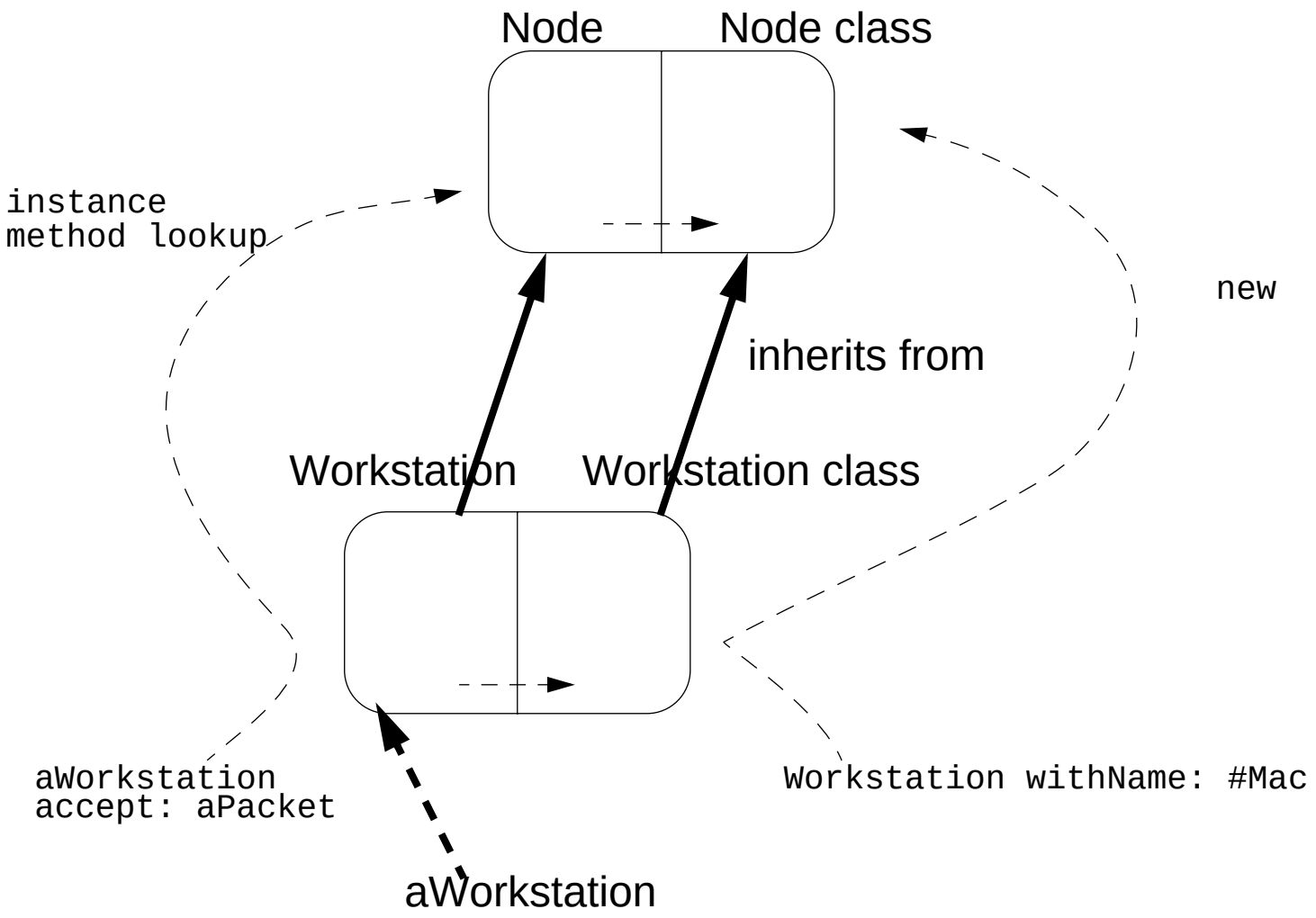
Let us be Absurb!

Let us suppose the WRONG hypothesis:
"IF super semantics =
starting the lookup of method in the superclass of the receiver class"

What will happen for the following message: aC m1
m1 is not defined in C
m1 is found in B
By Hypothesis: super = lookup in the superclass of the receiver class.
And we know that the superclass of the receiver class = B

=> That's loop So Hypothesis is WRONG !!

Lookup and Class Messages



Object Instantiation

Objects can created by:

- Direct Instance creation: `(basic)new/new:`
- Messages to Instances that Create Other Objects
- Class specific Instantiation Messages

Direct Instance Creation: (basic)new/new:

- #new/basicNew and new:/basicNew: are class methods

- 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: to precise the size of the created instance (indexed variable)

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

- #new/#new: can be specialized ot have a customized creation
- #basicNew/#basicNew: should never be specialized

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
```

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 reason:

arrays of bytes (like C string) are faster than storing an array of pointers, each pointing to a single byte.

Types of Classes

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

Related Method to class types: `#isPointers, #isBits, #isBytes, #isFixed, #isVariable, #kindOfSubclass`

- classes defined using `#subclass:` support any kind of subclasses

- classes defined using `#variableSubclass:` support only: `variableSubclass:` or `variableByteSubclass:`subclasses

- classes defined using `#variableByteSubclass`
  - can only be defined if the superclass has no defined instance variables
  - pointer classes anmd byet classes don't mix
  - only byte subclasses

Indexed Classes

- For class that needs 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 Class/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

What you should know

- Defining a class
- Defining methods
- Semantics of `self`
- Semantics of `super`
- Instance creation

Again open a browser and test!

6. Basic Elements of Design and Class Behavior

- Class definition
- Supporting Instance initialisation
- Supporting Instance creation

- Instance/Class methods
- Instance variable/ Class instance variables
- Class initialisation
- Abstract Classes

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: '; print: self addressee.
    aStream nextPutAll: ' with contents: '; print: 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’
```

Problems of this approach:

- responsibility of the creation relies on the clients
- can create packet without contents, without address
- instance variable not initialized -> error (for example, `printOn:`)
  - > system fragile

Assuring Instance Variable Initialization

Problem. By default #new class method returns instance with uninitialized instance variables. Remind that class methods cannot access to instance variables of an instance.
-> How to initialize a newly created instance ?

Moreover, #initialize method is not automatically called by creation methods #new/new:

Solution. Defines 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.
           contents := 'default message'

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

Remind. You cannot access instance variable from a class method like #new

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
```

Strengthen Instance Creation Interface

Problem. A client can still create a`Packet` 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
```

Problem! #new should raise an error!

The solution:

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

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

Class Methods - Class Instance Variables

- Classes (`Packet class`) represents class (`Packet`).
- Class instance variable 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
=> aNode hasNextNode

Node>>hasNextNode
   ^ self nextNode = self class noNextNode

Node class>>noNextNode
   ^ nil
```

Singleton Instance: A Class Behavior

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 returned it as result of #new.

```
|db|
db := LAN new.

db == LAN new -> true
LAN uniqueInstance == LAN new -> true
```

Singleton Instance's Implementation

```
LAN class
   instanceVariableNames: 'uniqueInstance '

LAN class>>new
   self error: ‘should use uniqueInstance’

LAN class>>uniqueInstance
   uniqueInstance isNil ifTrue: [ uniqueInstance := self basicNew initialize].
   ^uniqueInstance
```

Providing access to the unique instance is not always necesssary. It depends what we want to express. The difference between #new and #uniqueInstance is:

- #new potentially initializes a new instance.
- #uniqueInstance only returns the unique instance there is no initialization.

(see Smalltalk Companion for an interesting discussion)

Class Initialization

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

Example: `Date`

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

Date class>>initialize

```
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. "
   "Date initialize."

   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 )
```

Abstract Classes

- Should not be instantiated (deferred class of Eiffel).
- 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).

- Deffered method send the message `self subclassResponsibility`.
- Depending of the situation, override #new to produce an error.

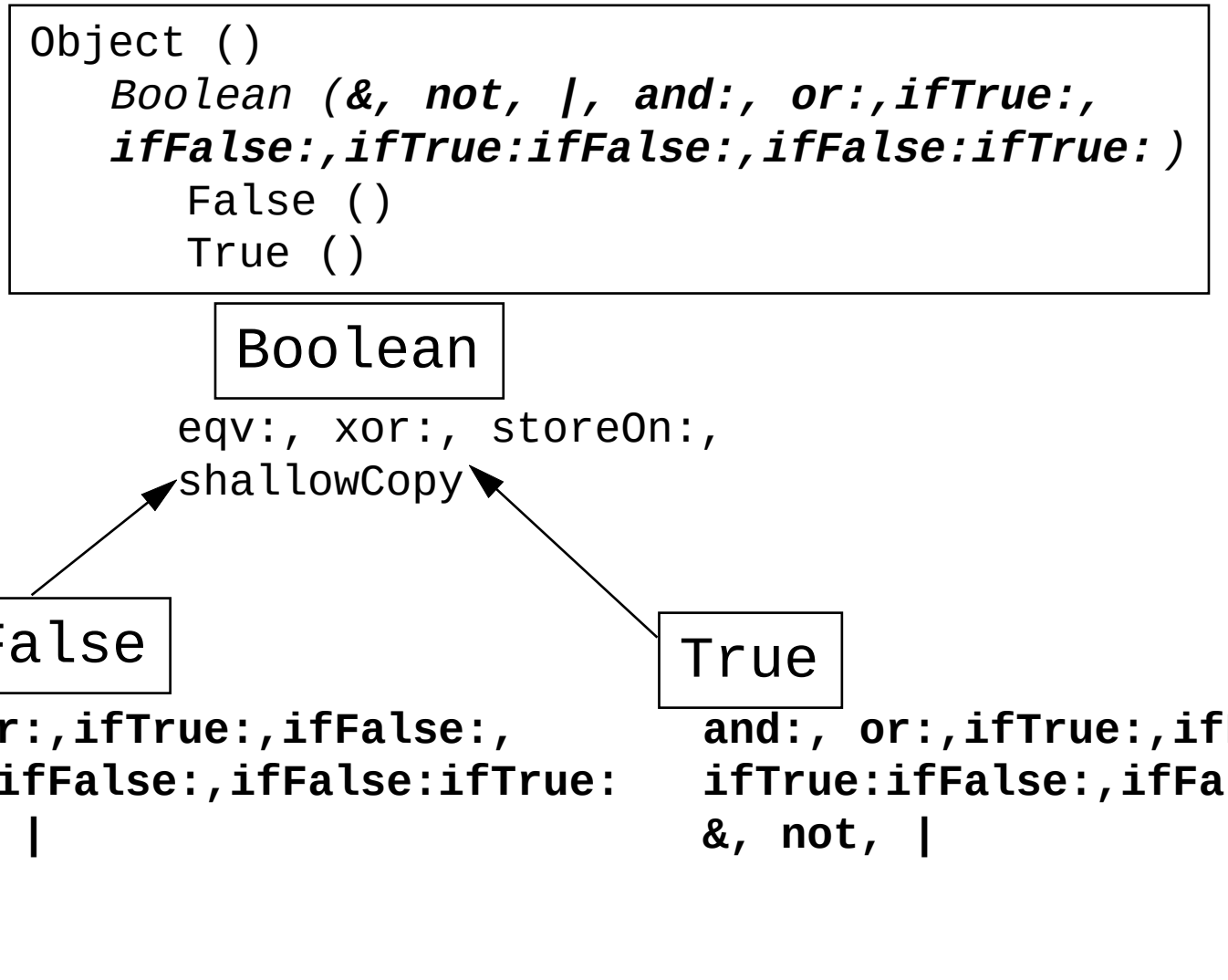
- Abstract classes are not syntactically distinguable from instantiable classes.

BUT as conventions 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.

```
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



Boolean

Abstract method

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

Concrete method efined 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, optimisation for essential methods and Just In Time compilation. A method is executed once and after it is compiled in native code. So the second time it is invoked the native code is executed.

False and True

```
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
```

CaseStudy: 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
```

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
```

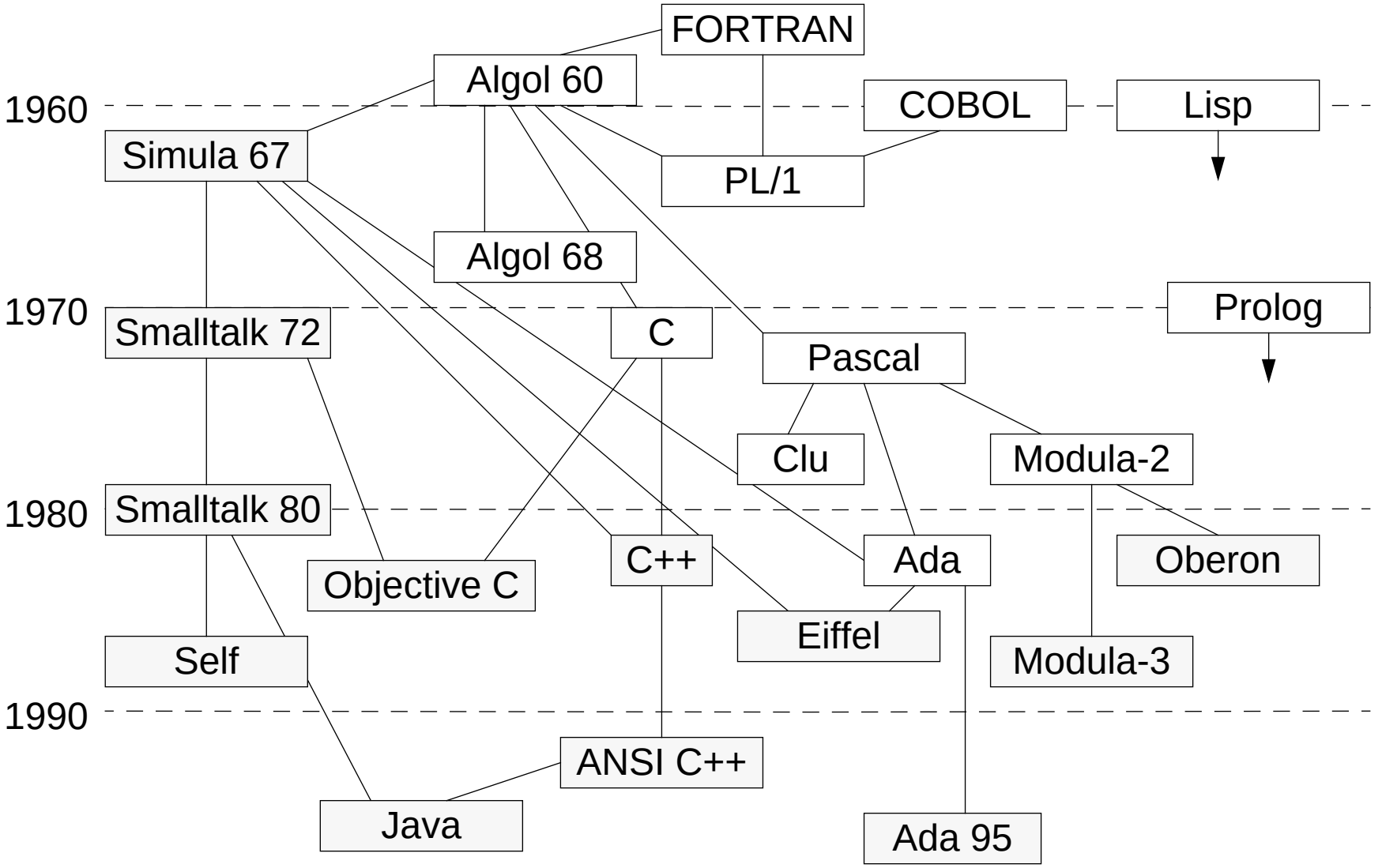
7. Comparing C++, Java and Smalltalk

Commented version

Overview

- History:
  - target applications, evolution, design goals
- Language features:
  - syntax, semantics, implementation technology
- Pragmatics:
  - portability, interoperability, environments & tools, development styles

History


FORTRAN
Algol 60
COBOL
Lisp
1960
Simula 67
PL/1
Algol 68
1970
Prolog
Smalltalk 72
C
Pascal
Clu
Modula-2
1980
Smalltalk 80
C++
Ada
Oberon
Objective C
Eiffel
Self
Modula-3
1990
ANSI C++
Java
Ada 95

Target Application Domains

Smalltalk

Originally conceived as PL for children.

Designed as language and environment for “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, 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

C++

- C with classes
  - ☞ and strong-typing, and ...
- "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 OO 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 (coming soon?)
type checking	dynamic	static	static
modules	no (categories)	no (header files)	packages
exceptions	yes (not commonly used)	yes (weakly integrated)	yes (well integrated)
concurrency	yes (semaphores)	no (libraries)	yes (monitors)
reflection	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 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!

— closed world assumption!

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

— *C++ now supports explicit name spaces? does this help?*

Java

- Static *and* dynamic type-checking (safe downcasting)
- Classes live inside packages

Exceptions, Concurrency

Smalltalk

- Can signal/catch exceptions — *seems not to be widely used!*
- 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* — *i.e. inspecting and reacting on an object's interface*

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 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

— *NB: Envy supports programming by teams (version control etc.)*

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)

— Not so great for intensive data processing, or client-side internet programming

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)

8. The Model View Controller Paradigm

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

Question:

- ➪ How can we reduce the complexity of the development of such applications

Answer:

- ➪ 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 fullfill.

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:
We structure our application according to the following partition:

- Model
- View
- Controller

Separation of Concerns I:

Functionality vs. User Unterface

Model:

- Domain specific information
- Core functionality, where the computation/data processing takes place

User Interface:

- Presentation of the data in various formats
- dealing with user input (Mouse, Keyboard, etc.)

Separation of Concerns II:

Display vs. Interaction

View:

- displaying the data from the model

Controller:

- relaying the user input to the View (e.g. Scrolling) or the model (e.g. modification of the data)

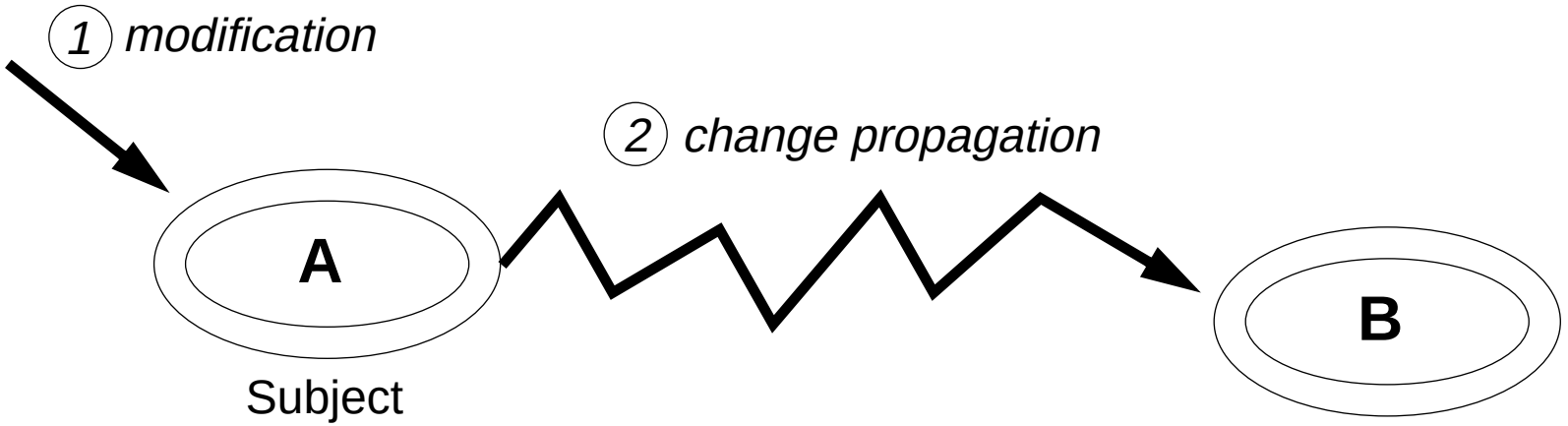
View and Controller are very much related. There is always a 1:1 relationship between views and controllers. There are examples of systems where view and controller are not separated.

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 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

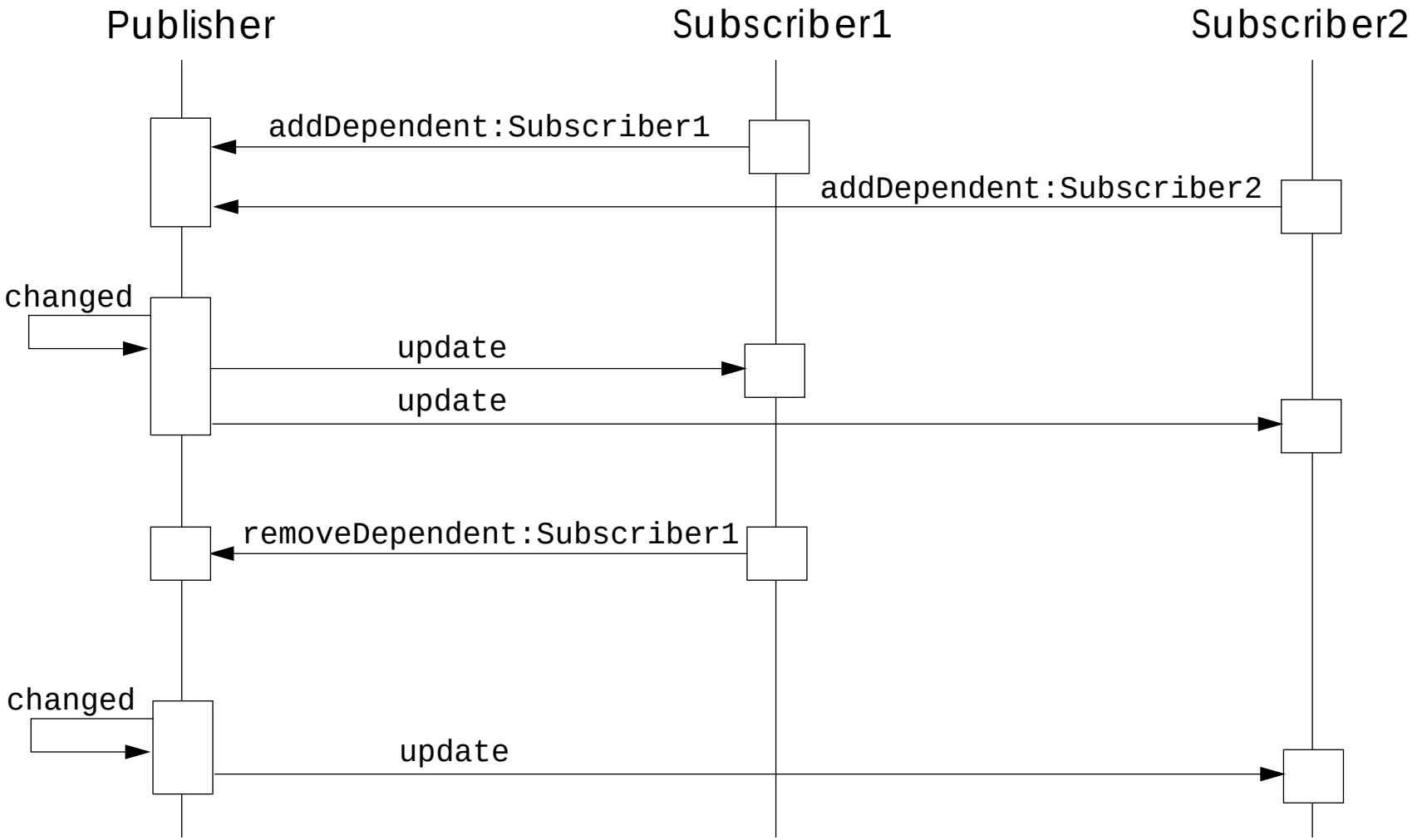
- adding and removing dependents
- change propagation

The publisher (subject) has a list of subscribers (observers, dependents). A subscriber registers with a publisher.

Protocol:

1. a publisher receives a `changed` message
2. all the subscribers receive `update` messages

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 enhances coupling between publisher and subscriber, or many a subscriber receives unnecessary data.

- **Pull**: the subscriber after receiving the update message asks the publisher for the specific data he is interested in

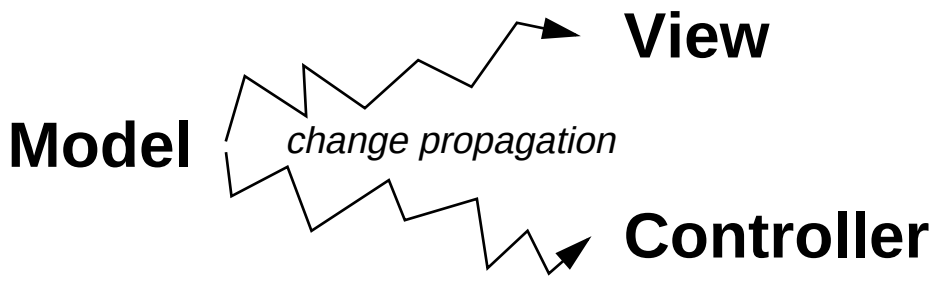
  Advantage: Only the necessary amount of data is transferred.

  Disadvantage: a lot of messages have to be exchanged.

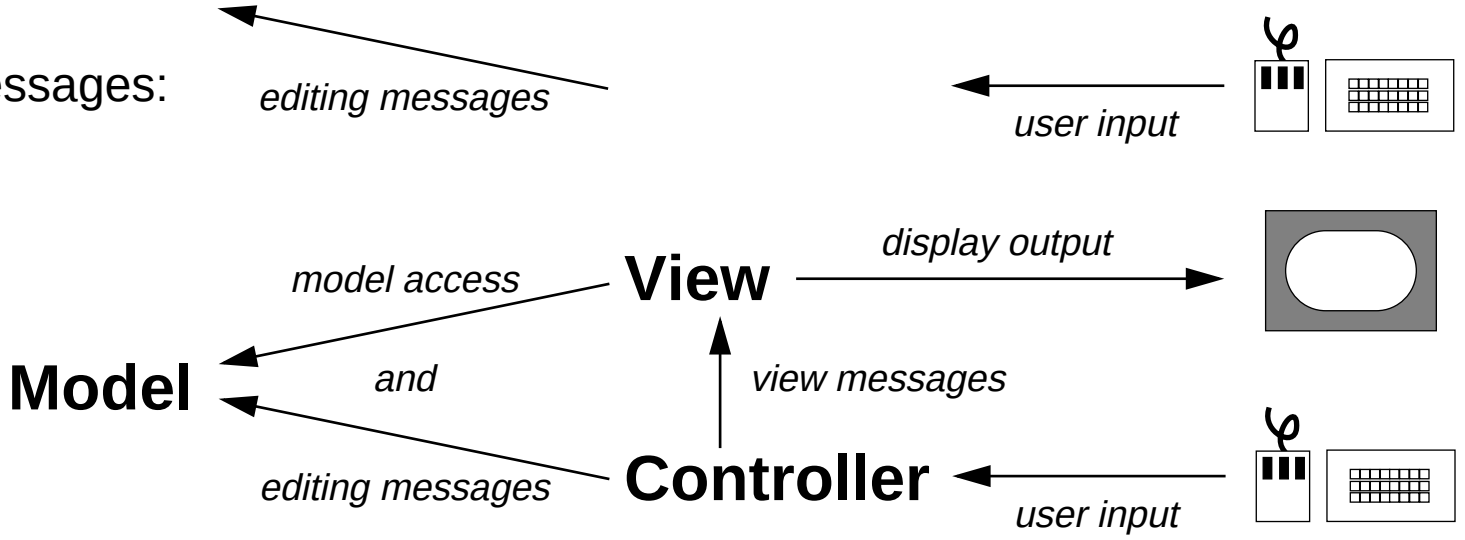
- Mixture: the publisher sends hints (“Aspects” in ST terminology) and other parameters along with the update messages

The MVC Pattern

Dependencies:



Other Messages:



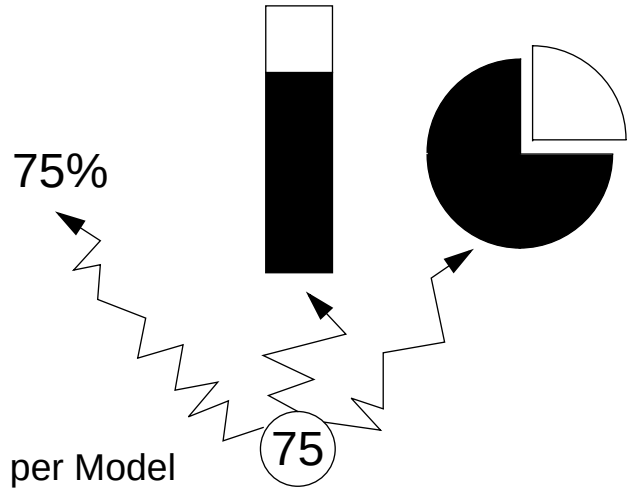
A Standard Interaction Cycle

<<diagram from the Buschmann et. al. book>>

MVC: Benefits and Liabilities

Benefits:

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



Multiple Views per Model

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 used to desing applictions independently of the programming language.

Examples:

- ET++ User Interface Framework (C++)
- Swing-Toolkit in the Java Foundation Classes 1.0 (due mid February 98)

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.

Managment of Dependents

Protocol to manage dependents (defined in `Object>>dependents access`):

- `addDependent:` anObject
- `removeDependent:` anObject

Attention: Storage of Dependents !

- `Object`: keeps the all his 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

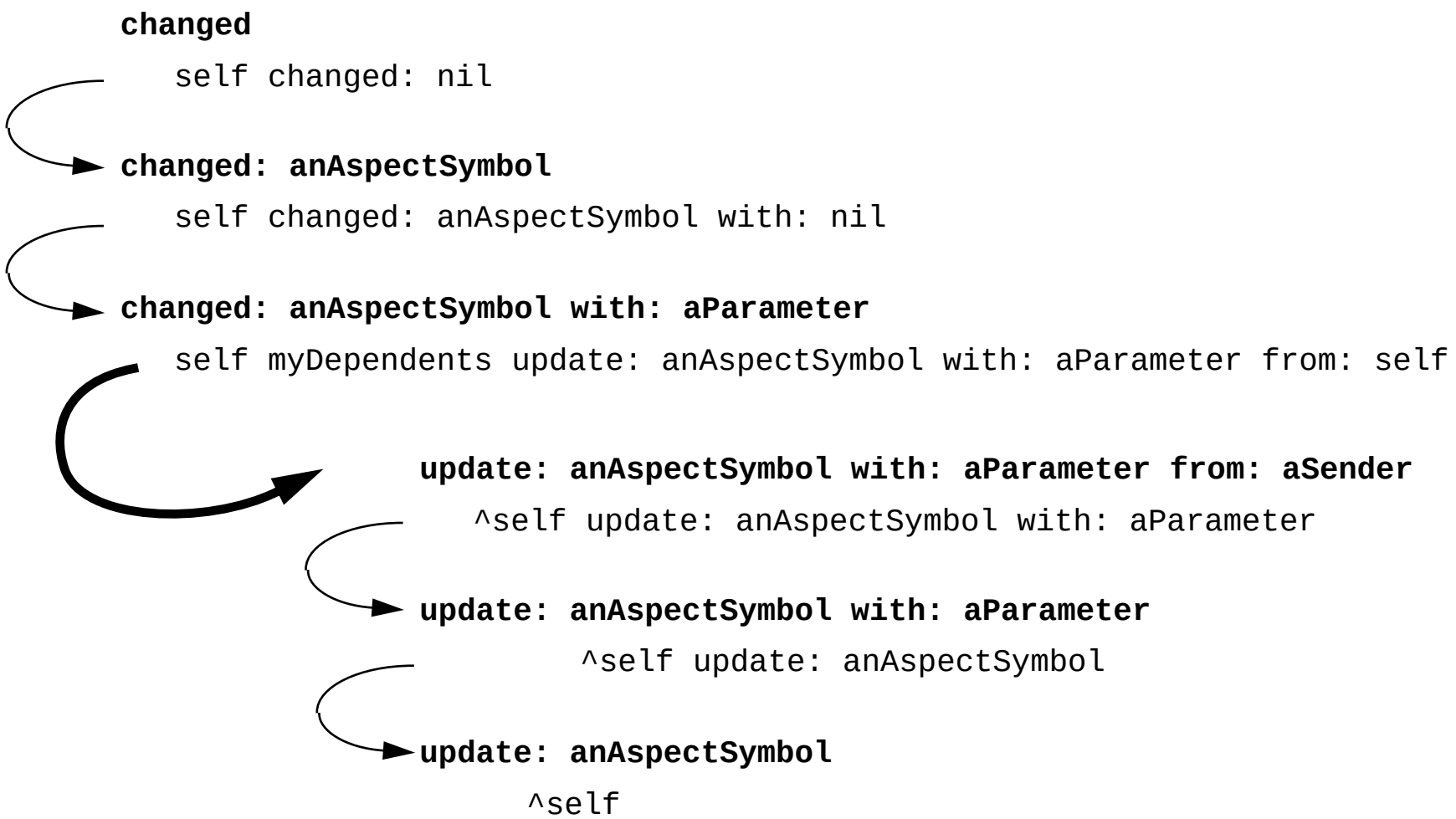
Change methods are implemented in `Object>>changing`:

```
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 information. 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



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:

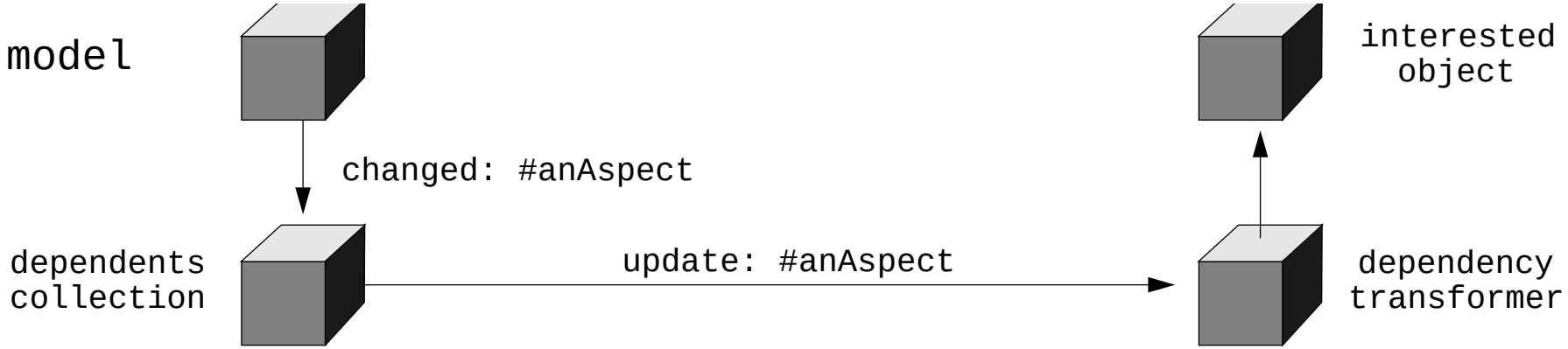
- publisher: send messages only to interested dependents
- subscriber: being notified directly by a call to the method that handles **that** specific change

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` at his 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`:

```
ValueHolder with: anObject
```

or

```
anObject asValue
```

Accessing a `ValueHolder`:

```
aValueholder value
```

```
aValueholder value: aNewValue
```

(notifies the dependents automatically)

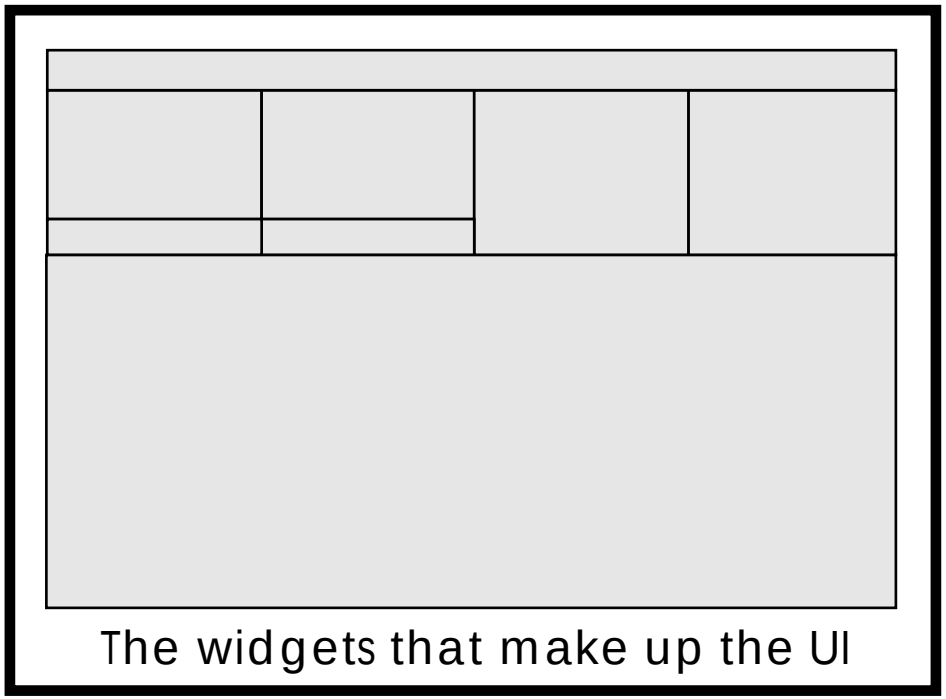
```
aValueholder setValue: aNewValue
```

(without notifying the dependents)

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



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:

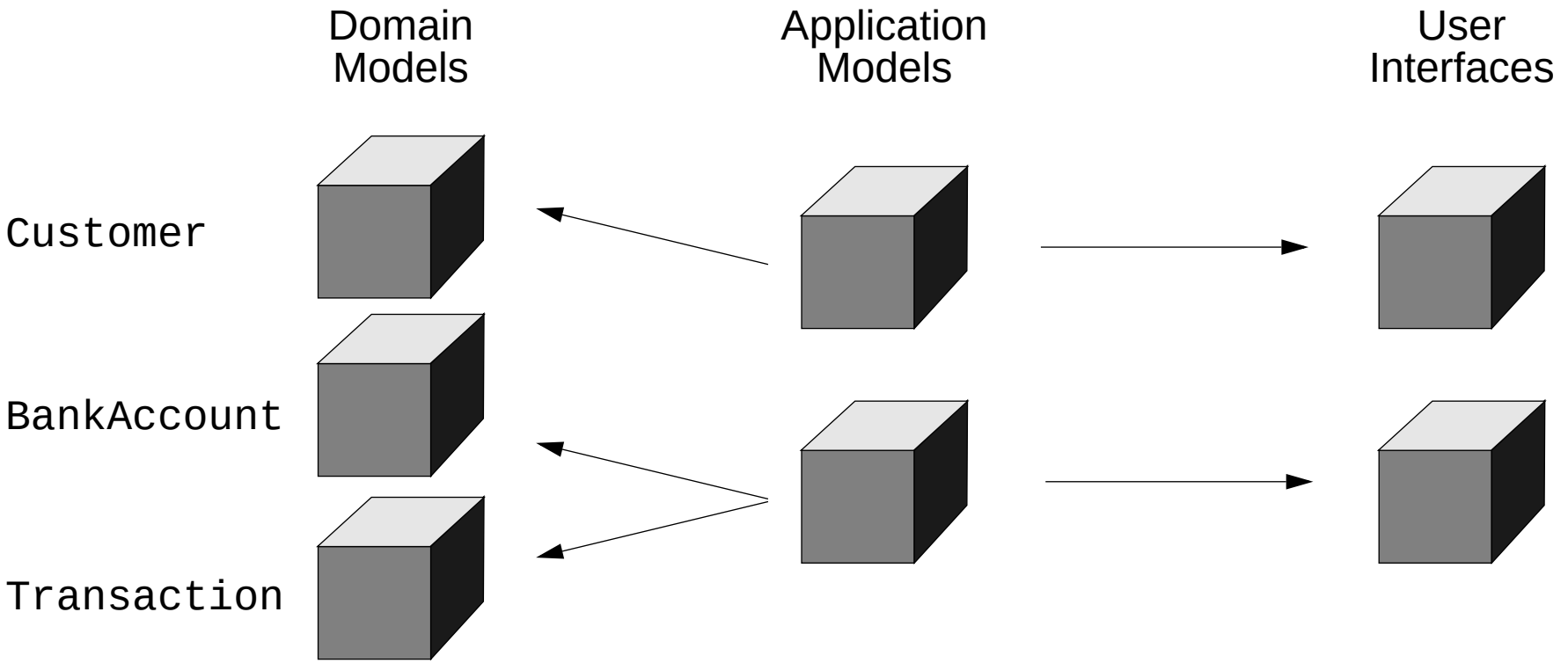
Value Model Protocol

Each model component is put into an *aspect model*, which can be a `ValueHolder` for example. The Widget deals only with this aspect model.

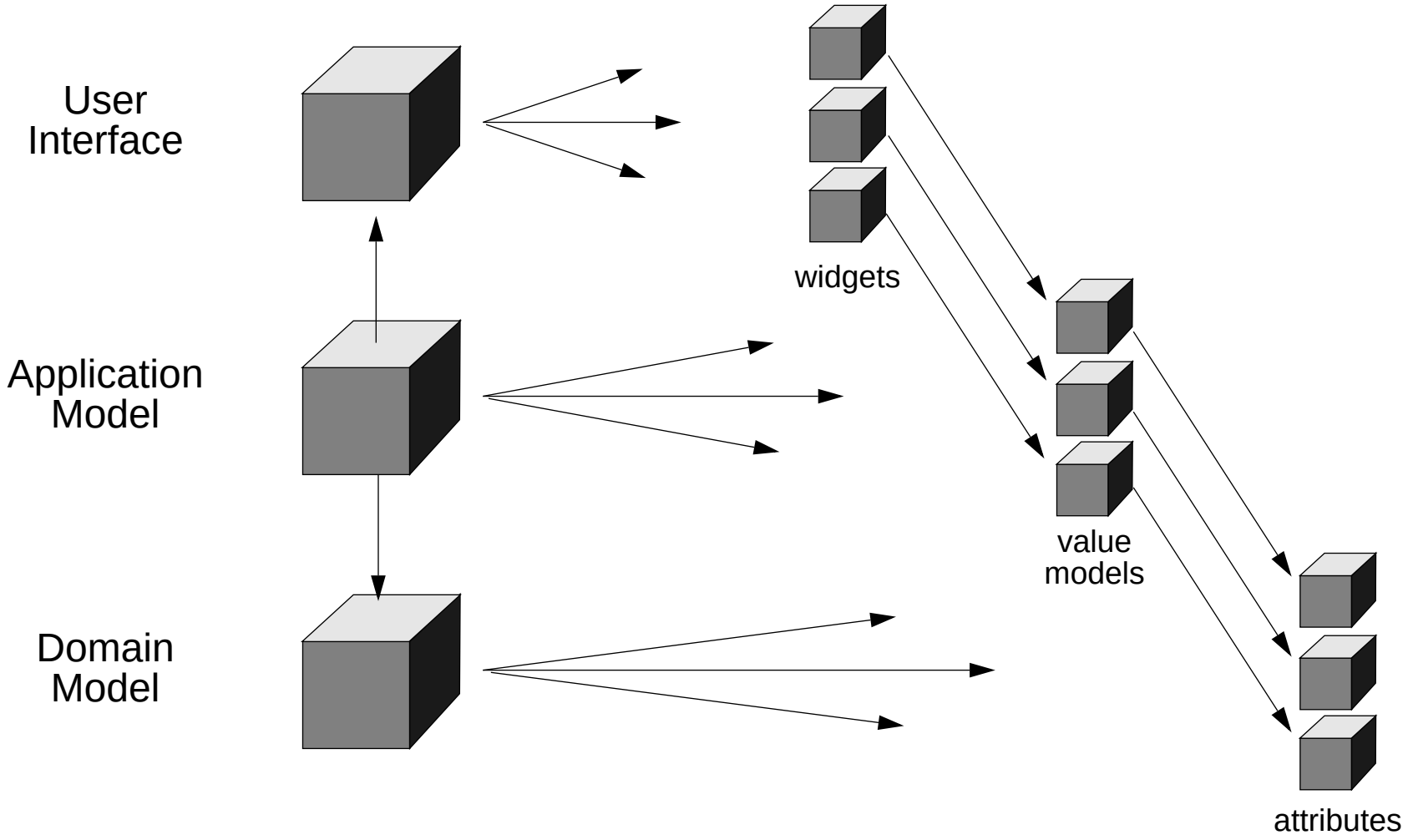
☞ the widget does not have to know any specifics about its model

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



MVC Bibliography

The Pattern:

E. Gamma et. al.: *Design Patterns*, Addison Wesley, 1995

- ☞ Observer, p. 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

9. Design Thoughts and Selected Idioms

The Object Manifesto

Be lazy:

- Never do the job that you can delegate to another one!

Be private:

- Never let someone else plays with your private data

The Programmer Manifesto

- Say something only once

About the Use of Accessors (i)

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

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

Schedule>>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 Public Accessors (ii)

This is not because there are methods in the interface that you provide a good data encapsulation.

If they are mentioned (no inforcement in Smalltalk) as public 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 a dictionary THAT'S BREAK.

So take care about the coupling between your objects and provide a good interface!

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

Returns consistenly the receiver or the element but the not the collection (else people can look inside and modifies it)

Composed Method

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 have the parameters of a Constructor Method to the class, how to you pass them to the newly created instance?

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

But violates the “say things only 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]
```

Defines

```
    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

But also we can use `sortBlock:` of `SortedCollection class`

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

Execute Around Method

How do 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

Cursor>>showWhile: aBlock
    |old|
    old := Cursor currentCursor.
    self show.
    aBlock value.
    old 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

What 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 implementations.

Name Well your Methods (i)

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:

Instead of writing that:

```
|index|
index := 1.
[index <= aCollection size] whileTrue:
   [... aCollection at: index...
   index := index + 1]
```

Write that

```
aCollection do: [:each | ...each ...]
```

collect:

Instead of :

```
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 that:

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

Note that this solution works well for indexable collection and also for sets.
The previous one not!!!

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]].
...
```

Or:

```
|found|
found := (aCollection
       detect: [:each| each | anObject]
       ifNone:[ nil]) notNil.
```

Write:

```
|found|
found := aCollection includes: anObject
```

Class Naming

- 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
```

10. 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:

1. *Multiprogramming:* processes share one or more processors
2. *Multiprocessing:* each process runs on its own processor but with shared memory
3. *Distributed processing*: each process runs on its own processor connected by a network to others

Motivations for concurrent programming:

1. Parallelism for faster execution
2. Improving processor utilization
3. Sequential model inappropriate

Limitations

But 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 a`Process` is to send a block a 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

- 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.

```
aProcess resume
```

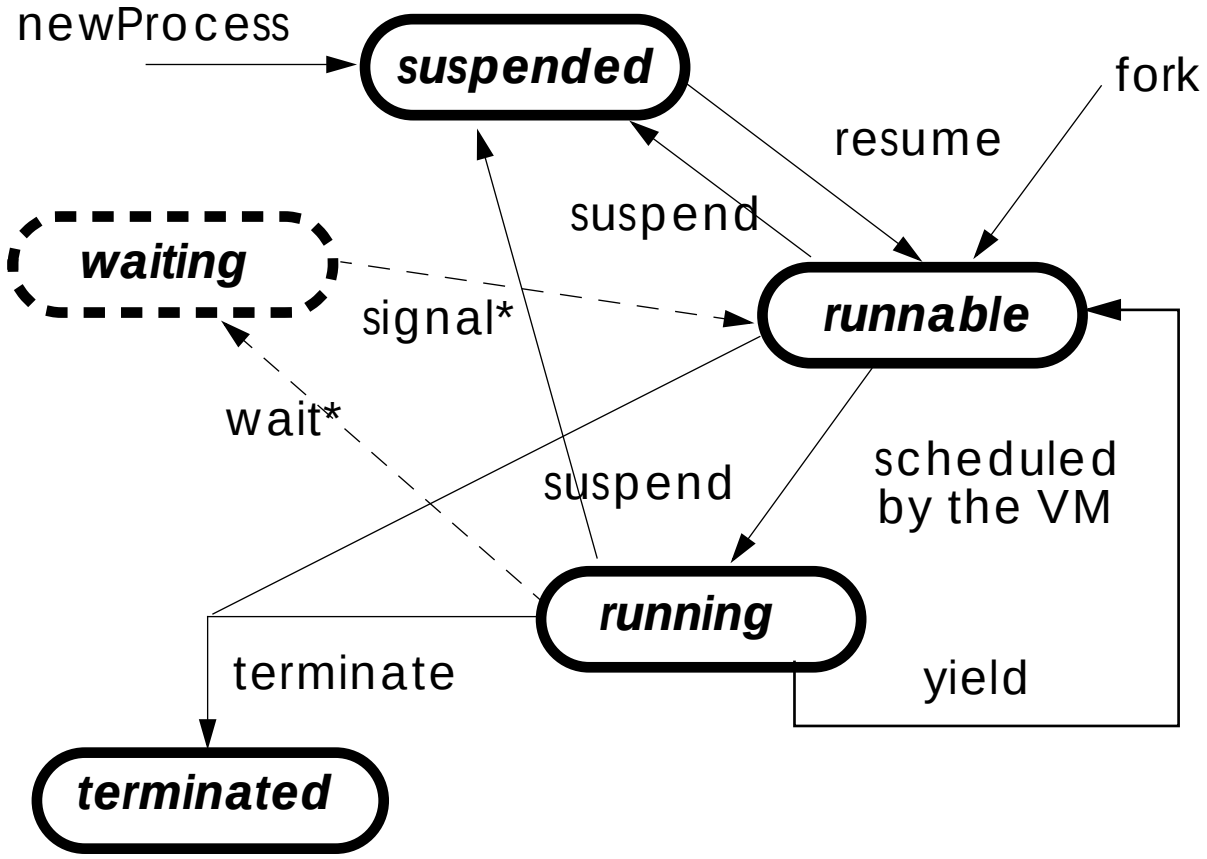
- A process can be created with any number of arguments:

```
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 dependant 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

Processes Scheduling and Priorities

- Process scheduling is done by the unique instance of class `ProcessorScheduler` called `Processor`.

- A runnable process can be created with an specific priority using the `#forkAt:` message:

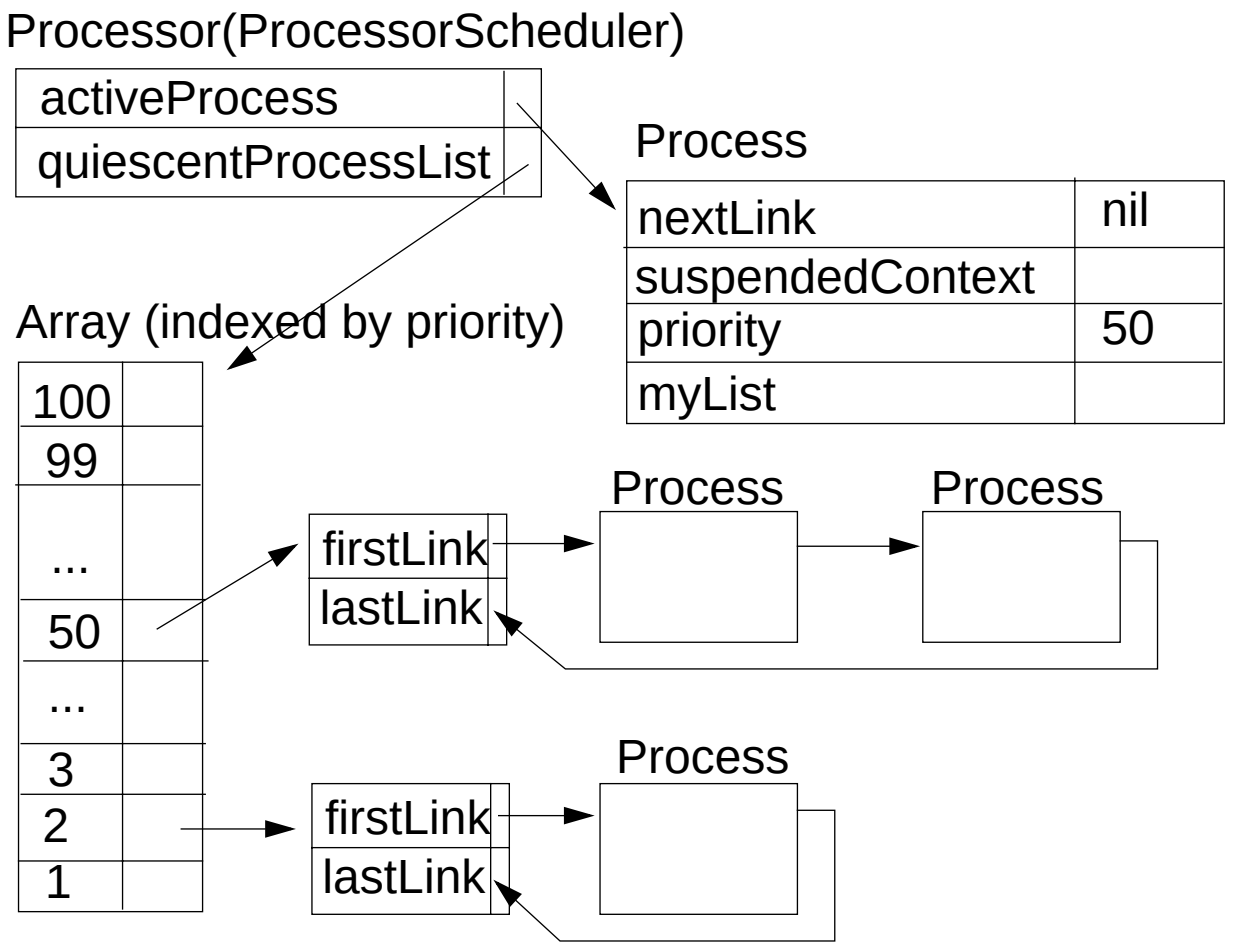
```
[ Transcript cr; show: 5 factorial printString ]
      forkAt: Processor userBackgroundPriority.
```

- 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)*

Processes Scheduling: The Algorithm



-The active process can be identified by the expression:

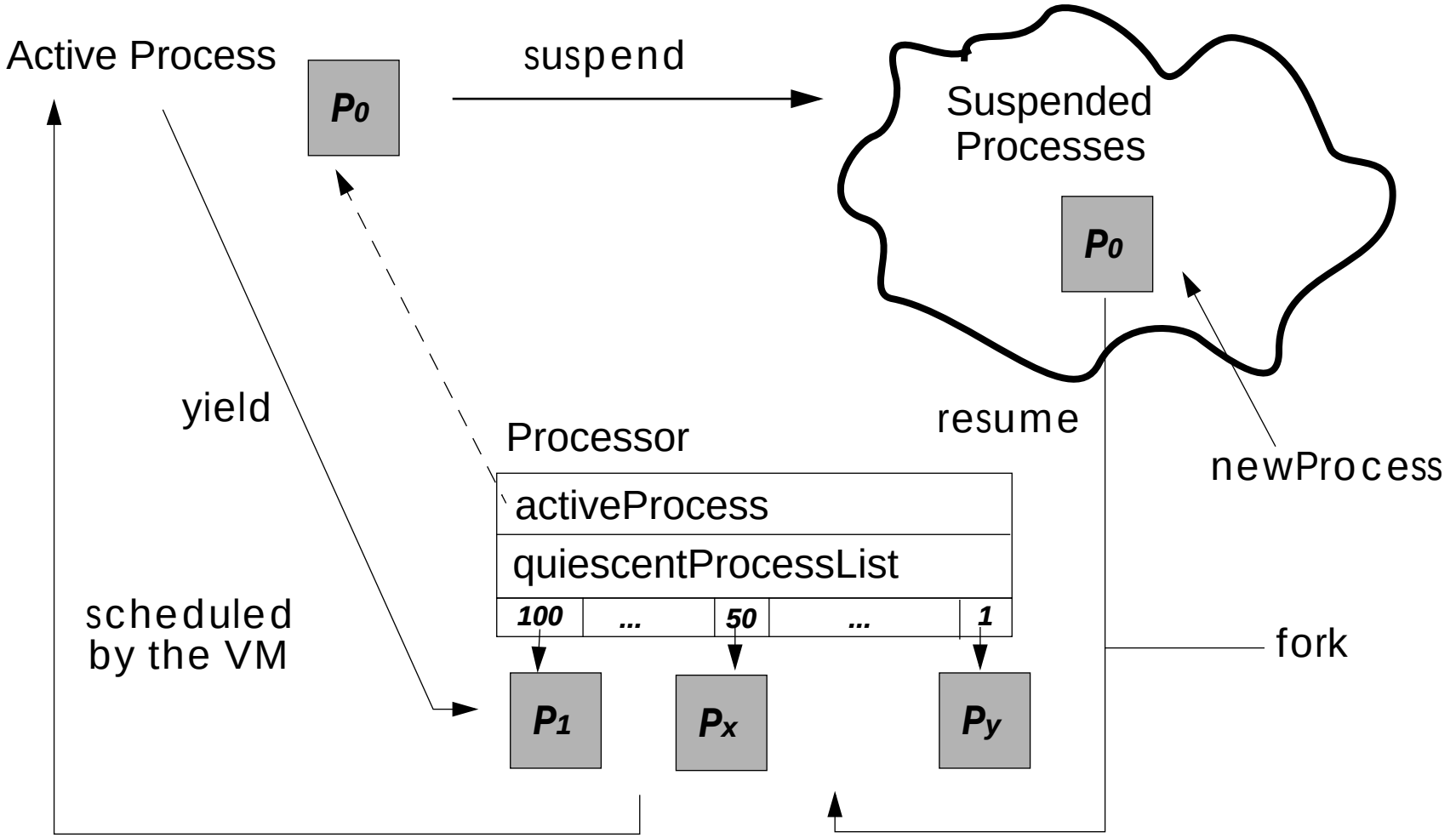
```
Processor activeProcess
```

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

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

-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

Processes have references to some common objects, such objects may receive messages from several processes in an arbitrary order. This may 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 is = ‘; show: temp printString ] forkAt: 60.

P1 running
P1 sum is = 5000050000
```

Synchronization Mechanisms

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
```

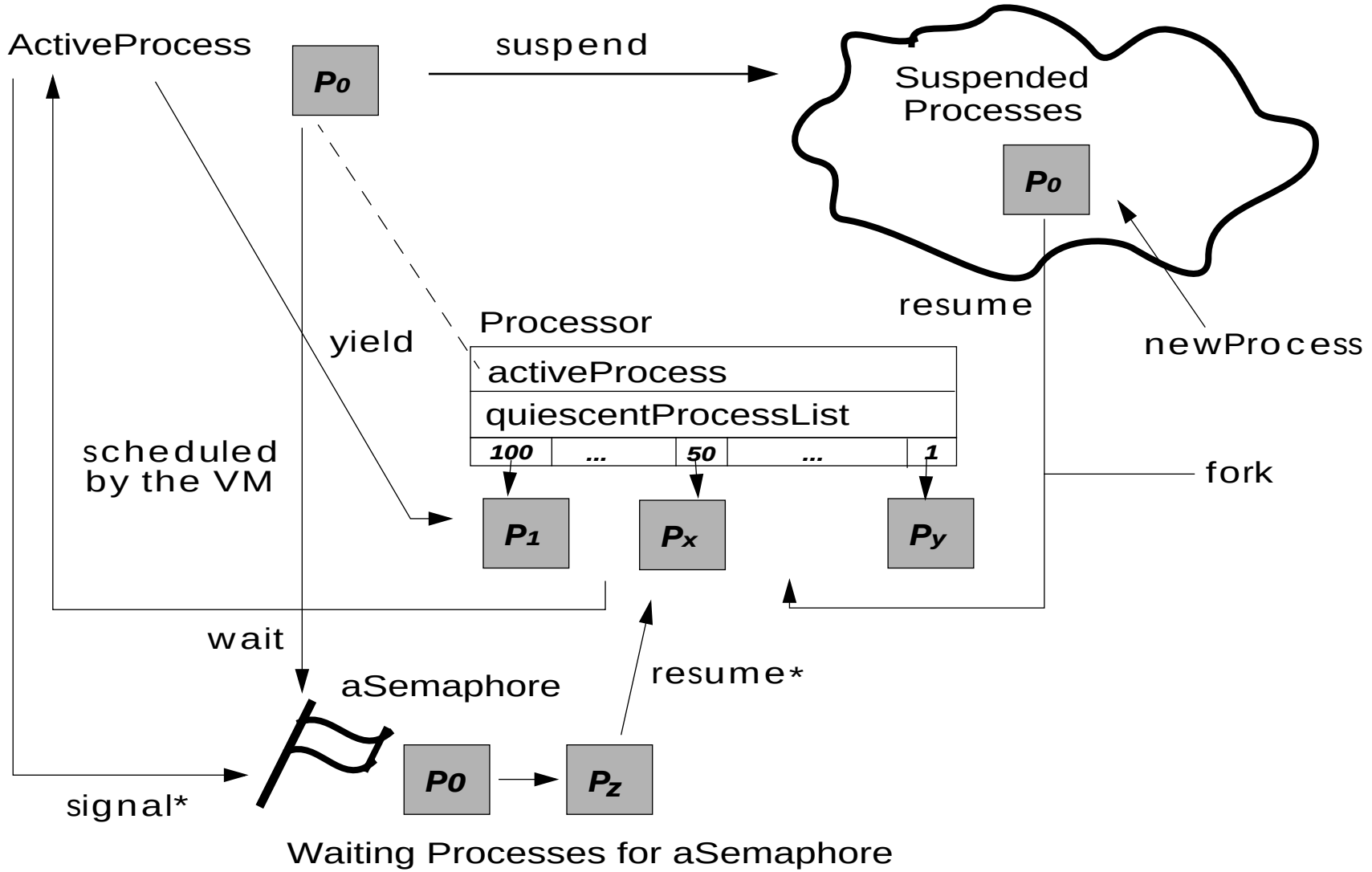
Synchronization using Semaphores

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


ActivePProcess
P0
suspend
Suspended
Processes
P0
resume
newProcess
yield
Processor
activeProcess
quiescentProcessList
100
...
50
...
1
scheduled
by the VM
fork
P1
Px
Py
wait
aSemaphore
resume*
P0
Pz
signal*
Waiting Processes for aSemaphore

Semaphores for Mutual Exclusion

A semaphore is used frequently to provide mutual exclusion from a “critical region”. 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 out 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 to synchronize communication between processes. Its works like a normal queue (First in First Out, reads and writes), the main difference is that aSharedQueue protects itself against possible concurrent access (multiple writes and/ or multiple reads).

Processes add objects to the sharedqueue by using the message #nextPut: (1) and read objects from the sharedqueue 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 into the sharedqueue when the messsage #next is received, the process is *suspended.*
- We can request if the sharedqueue is empty or not by using the message #isEmpty

Delays

Instances of class Delay are used to cause a real time delay in 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 for resumption of the active process 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 cannot be sent the message wait repeatedly.

Promises

- Class `Promise` provides a means of evaluating a block in 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 process has completed by sending the message `#hasValue`

11. Classes and Metaclasses: an Analysis

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

At first look, a difficult topic!
You can live without really understand them
But metaclasses give a uniform model and you will made less errors.
And you will really understand the Smalltalk model

Recap on Instantiation
Recap on Inheritance

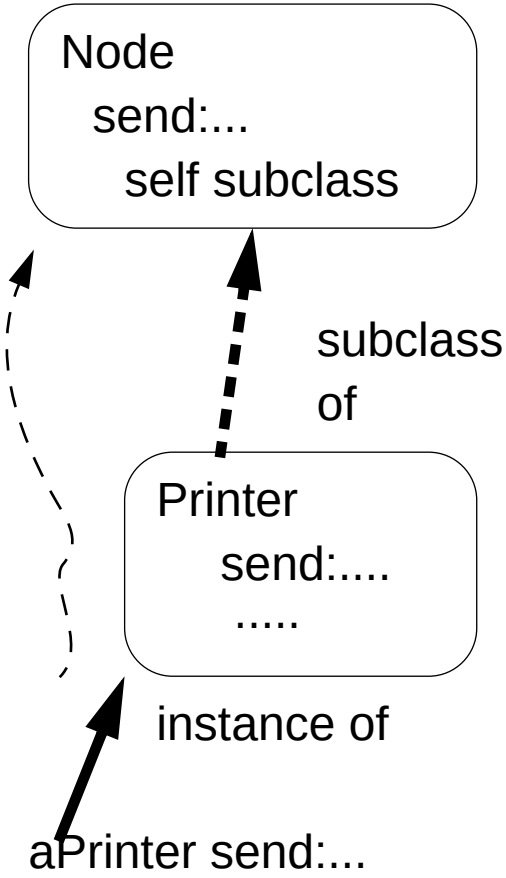
The meaning of "Instance of"

- Every object is an instance of a class.
- Every class is ultimately subclass of Object (except Object).
- When anObject receives a message, the method is lookup 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 with other instances the bevahior defined in its class via the instance of link.



Concept of Metaclass & Responsibilities

Concept:

- Everything is an object
- Each object is instance of one class
-A class is also an object instance of a metaclass
- 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, ...)

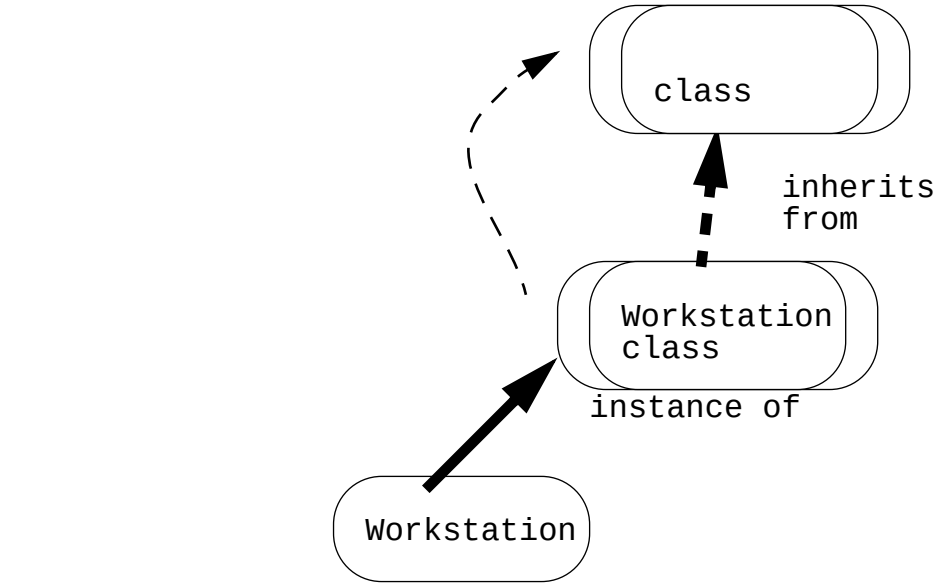
Examples:

```
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 lookup in its class and/or its superclasses.

So when aClass receives a message, the method is lookup 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)
- so all the classes should inherit ultimately from Object

    Workstation inherits from Node

    Node inherits from Object

Class

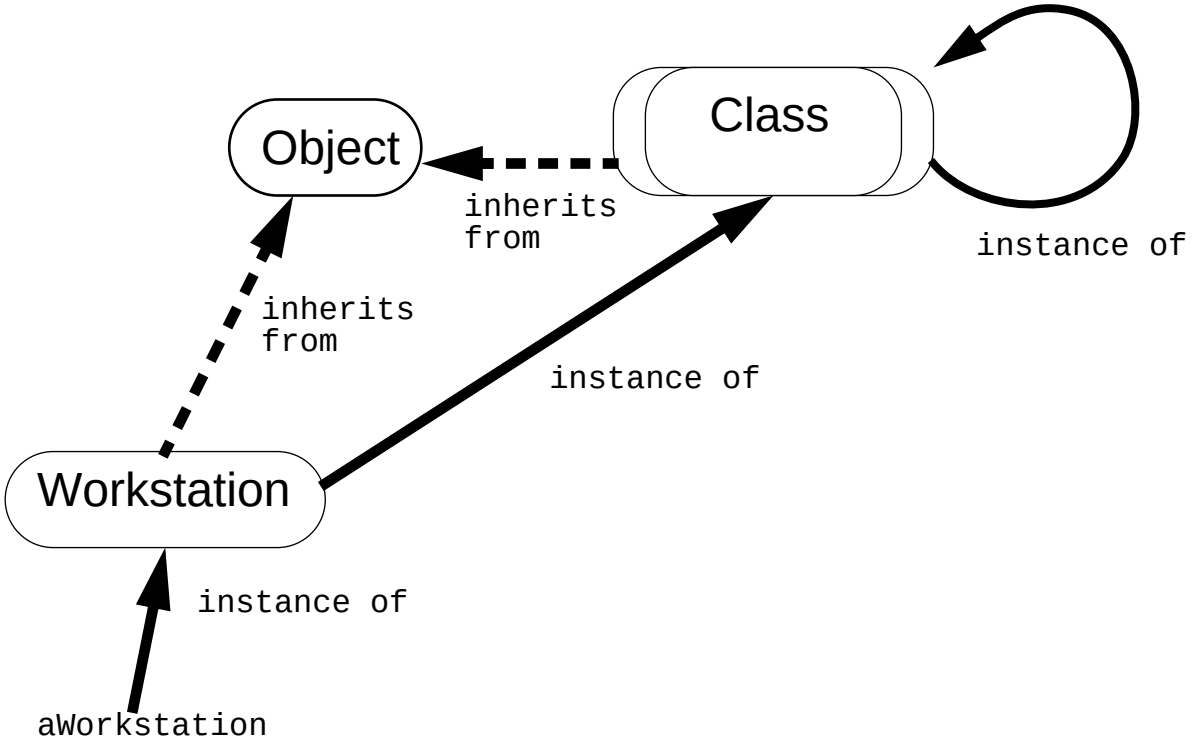
- represents the common behavior of all the classes (compilation, method storing, instance variable storing)
- Class inherits form Object because Class is an Object but a special one.

=> Class knows how to create instances

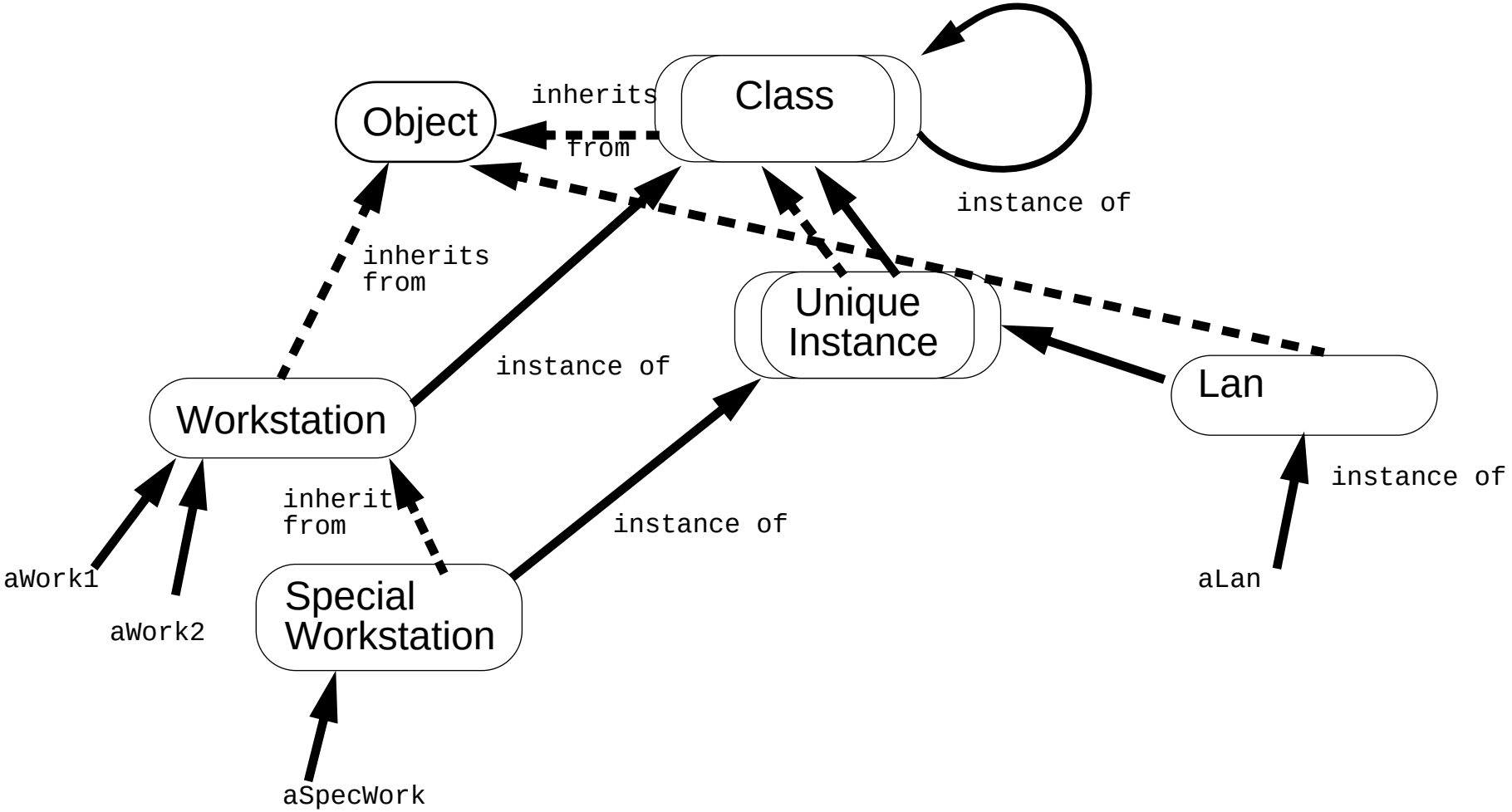
- So all the classes should inherit ultimately form Class

A possible kernel for explicit metaclasses

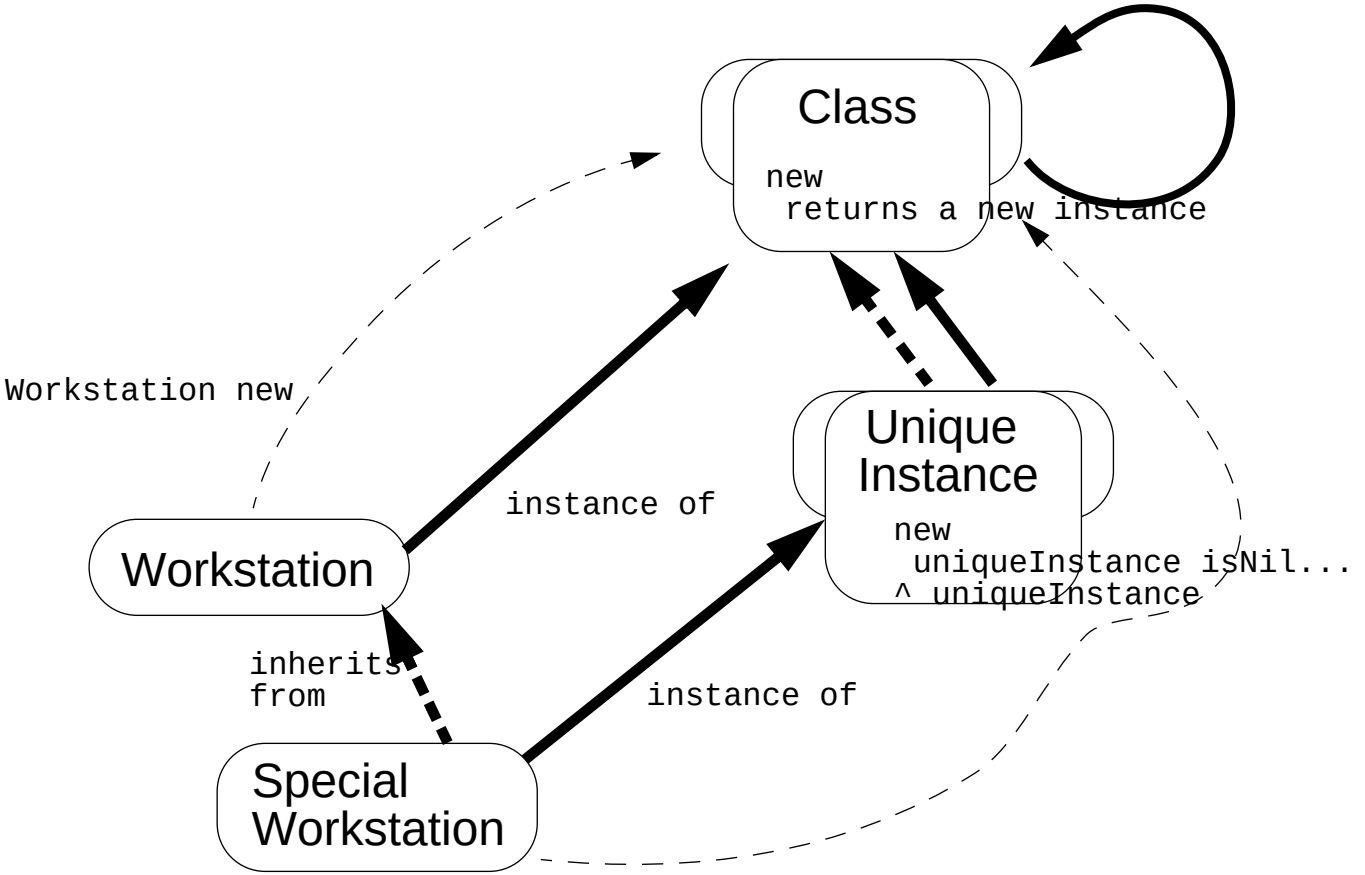
The kernel of CLOS and ObjVlisp but not the kernel of Smalltalk



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 instance of a class.

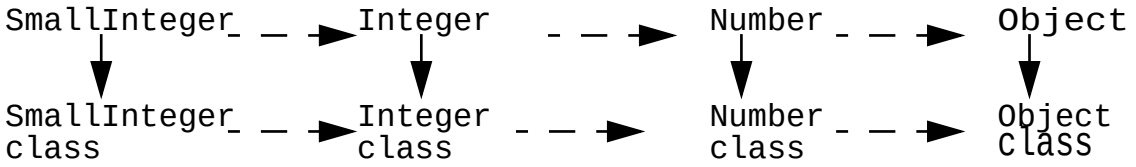
Each class is instacne of a class its metaclass.

(3) Every class is instance of A metaclass.

Every user defined class is the **sole** instance of another class (a metaclass).

Metaclass are system generated so they are unamed you can accessed them using `#class`

Smalltalk Metaclasses in 7 points (ii)

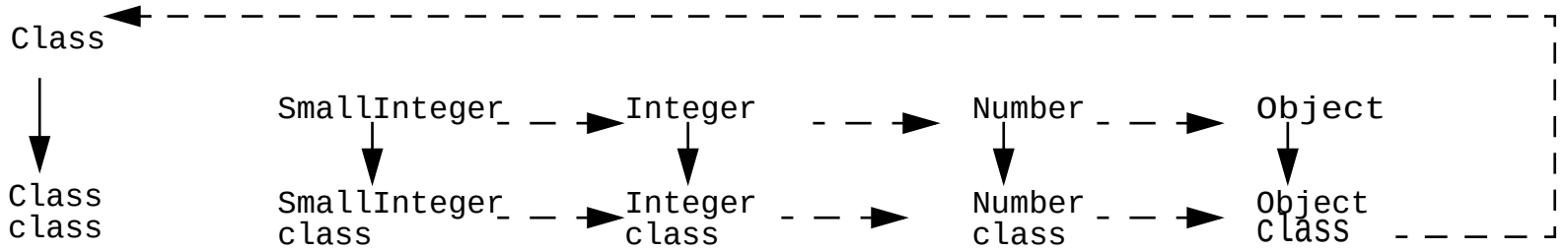


If X is a subclass of Y then X class is a subclass of Y class.

But what is the superclass of the metaclass of `Object`?

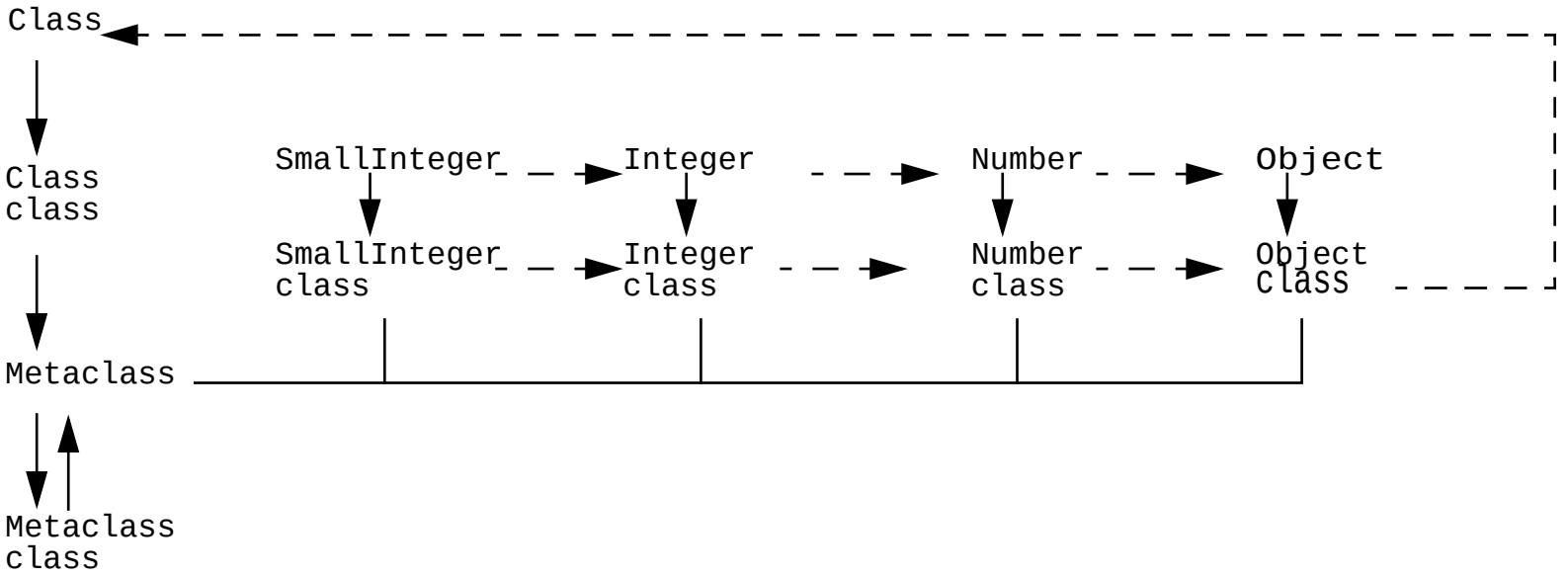
The superclass of `Object` class is `Class`

(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 instance of `Metaclass`. `Metaclass` is instance of itself

`Object` : common object behavior

`Class`: common class behavior (name, multiple instances)

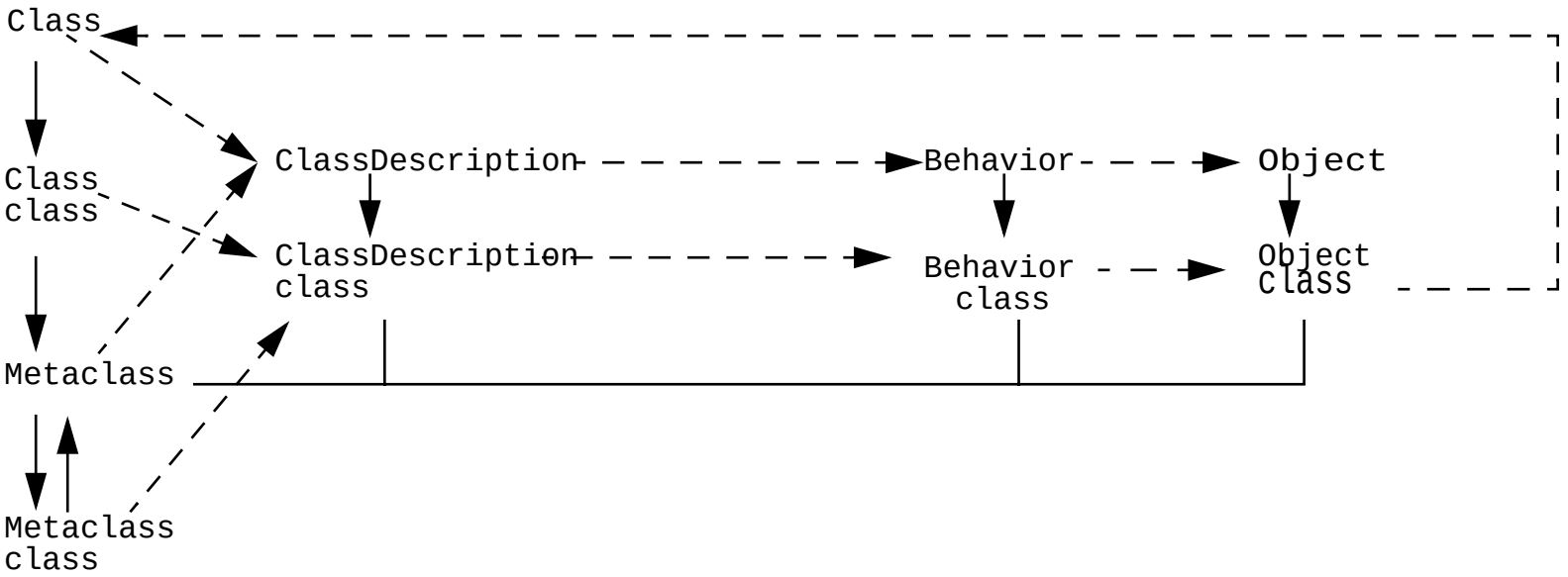
`Metaclass`: common metaclass behavior (no name, unique instance)

(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 mechanism for fileOut

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

Subclasses must implement

```
#addInstVarName:
#removeInstVarName:
```

Instance Variables:

- instanceVariables<Array of: String> names of instance fields
- organization <ClassOrganizer> provides organization of message protocol

Metaclass Responsibilities

- initialization of class variables
- creating initialized instances of the metaclass's sole instance

- instance creation (`#subclassOf:`)
- metaclass instance protocol (`#name:inEnvironment:subclassOf:`....)

Class Responsibilities

`Class` adds naming for class

`Class` adds the representation for classVariable names and shared pool variables (`#addClassVaraNames, #addSharedPool:, #initialize`)

12. Debugging

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

Most Common Beginner Bugs

- `true` is the boolean value, `True` its class

Instead of:

```
Book>>initialize
   inLibrary := True
```

that:

```
Book>>initialize
   inLibrary := true
```

- `nil` is not acceptable for `ifTrue:`

- `whileTrue` receiver must be a block

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

- (weakness of the system) Before creating a class check if it already exists

```
Object subclass: #View
```

- Do not assign class

```
OrderedCollection := 2 will damage your system
```

Return Value

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

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

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

returns `self` : the class `Packet` and not the newly created instance !!!

- Take care about loops

```
Book>>new
   ^self new initialize
```

Redefinition Bugs

- **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
```

- Before redefining new like `super new initialize` check if this is not already done. Else twice that expression in the same hierarchy will call twice `initialize`

Compile time errors

- Do not try to access **instance** variables to initialize them in the `#new` method.
You do not have the rigth.
Define and invoke `#initialize` method on instances.

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

- Do not try to assign a method argument

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

Library Behavior-based Bugs

- #add: returns the argument and not the receiver
So use yourself

- Do not forget to specialize #copyEmpty when adding named instance variables to a subclass having **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]
```

Copy first the collection

```
timers copy do:[:aTimer|
   aTimer isActive ifFalse: ‘timers remove: aTimer]
- Take care the iteration can involve different methods and can be less obvious!
```

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: usefull to add a id for debugging purpose

Where am I and how did I get here?

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
```

Source Inspection

Source Code for Blocks

```
aBlockClosure method getSource
aMethodContext sourceCode
```

Decompiling a Method

Shift + select the method is the browser

Interesting for literals modification or 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 is restored. So think also to return copy of your literals.

Entry Points

How a window is opened or what happens when the menu is invoked?

look into `LauncherView` and `UIVisualILauncher` implementors of '*enu*'

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: usefull for complex graphics

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

(Do not forget the wait)

Until a mouse button is cliked.

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

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’
ex:
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)
```

Finding & Closing Open Files in VW

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

How to ensure that an open file willl be close in case of 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."
```