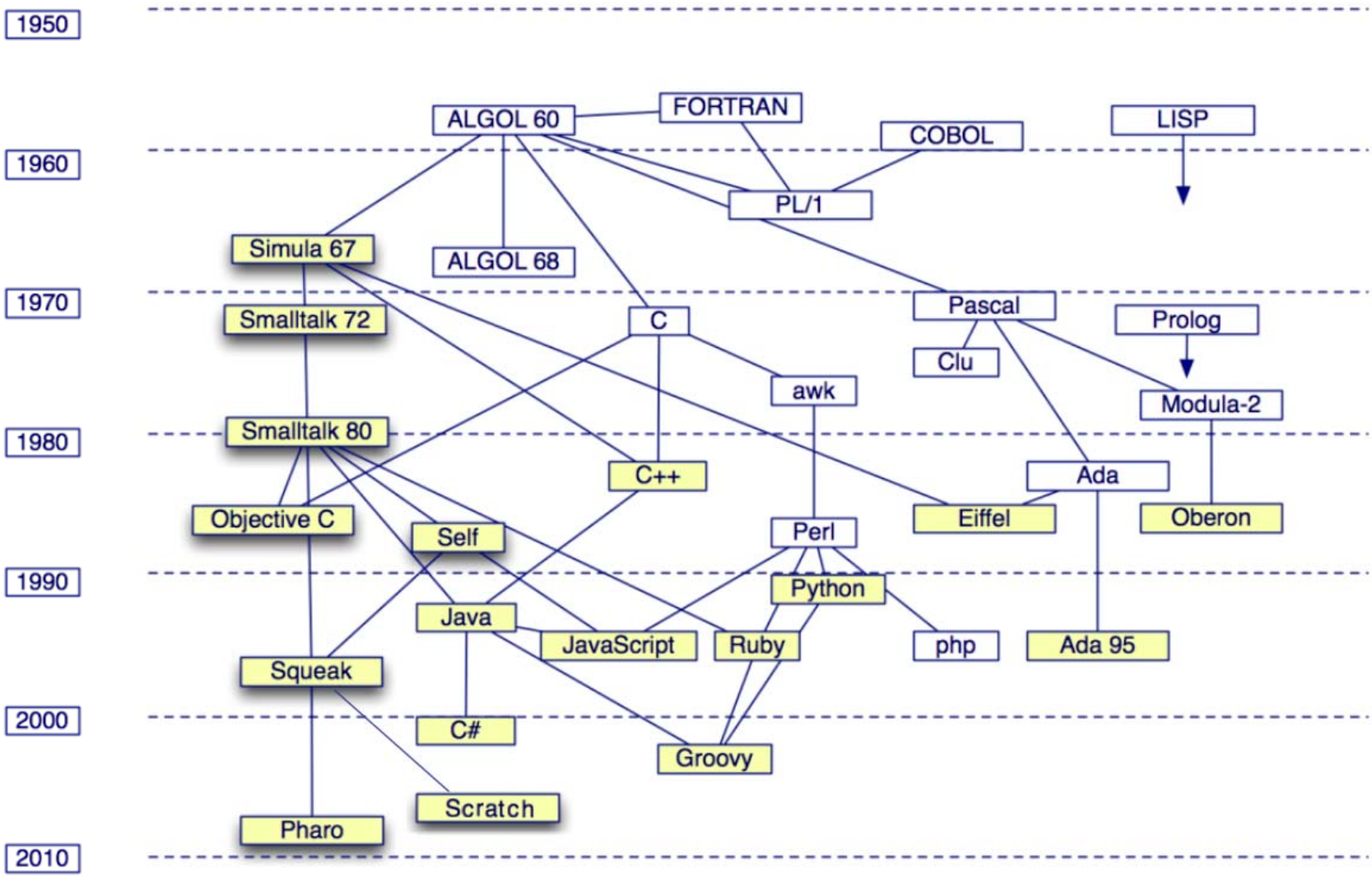
Smalltalk

School of Engineering

Based on Lecture
Prof. O. Nierstrasz
and
Pharo Online ProfStef
Tutorial

Influence Language



https://techbeacon.com/app-dev-testing/how-learning-smalltalk-can-make-you-better-developer

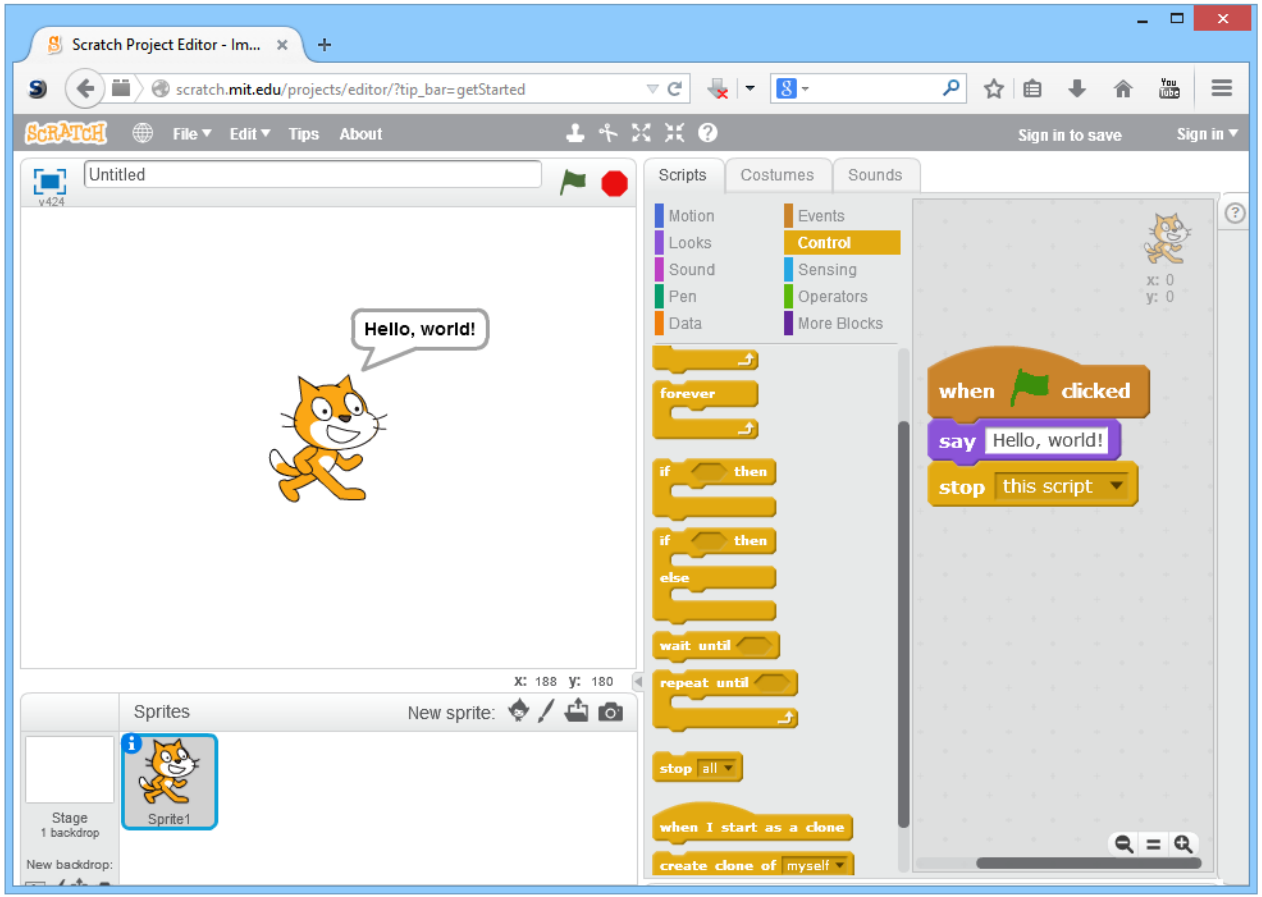
Smalltalk vs. C++ vs. Java

School of Engineering

	Smalltalk	***C++***	***Java***
Object model	Pure	Hybrid	Hybrid
Garbage collection	Automatic	Manual	Automatic
Inheritance	Single	Multiple	Single
Types	Dynamic	Static	Static
Reflection	Fully reflective	Introspection	Introspection
Concurrency	Semaphores, Monitors	Some libraries	Monitors
Modules	Categories, namespaces	Namespaces	Packages

Scratch for LP 21

- Was written originally in Smalltalk
- Language/Environment is Pharo/Squeek alike but more colorful



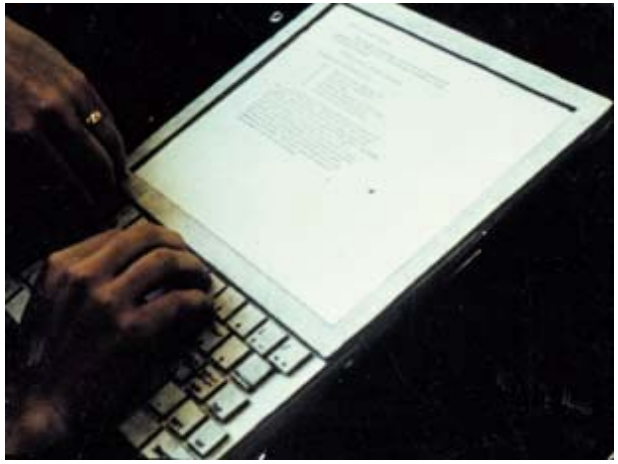
History

History Xerox PARC Internal Development

- **1972 — First Interpreter**
  - by Alan Kay & Dan Ingalls
  - A bet of Ingalls, that he could do it within a few pages of code
- **1976 — Redesign**
  - hierarchy of:single root
  - fixed syntax
  - compacter bytecode,
  - processes
  - semaphores
  - object/class browsers
  - GUI library.
- Projects: ThingLab, Visual Programming Environment

… History: Alan Kay's Dynabook Mockup

- Alan Kay: 1973 visionered a small portable computer
- At a time most computers looked like THIS !



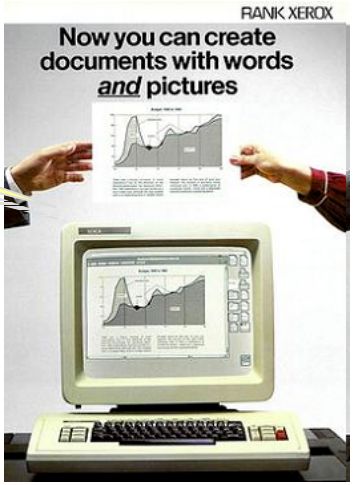
Dynabook Mockup

… History: Alto a Machine to Run Smalltalk

School of Engineering

- Developed at the research center of Xerox in 1973
- Alto the first **Workstation/PC**
  - Bit-mapped black and white display sized 606x808
  - as sheet of paper, aligned vertically
  - 5.8 MHz CPU.
  - 128KB of memory (at the cost of $4000)
  - 2.5MB removable cartridge hard drive.
  - Three button mouse.
  - 64-key keyboard and a 5-finger key set
- Commercialized as Xerox Star: failed

WYSIWYG



… History: Smalltalk Inventions

School of Engineering

- First to be based on Graphics
  - **Multi-Windowing Environment** (Overlapping Windows)
  - Integrated Development Environment: Debugger, Compiler, Text Editor, Browser
- With a pointing device -> *a **Mouse***
- Platform-independent **Virtual Machine**
  - **Just-in-time Compilation**
- **Garbage Collector**
- Everything was there, the complete Source Code

… History: Xerox PARC Inventions

- **Laser printers**
- Computer-generated **bitmap graphics**
- The **graphical user interface**, featuring windows and icons, operated with a mouse
- The **WYSIWYG** text editor

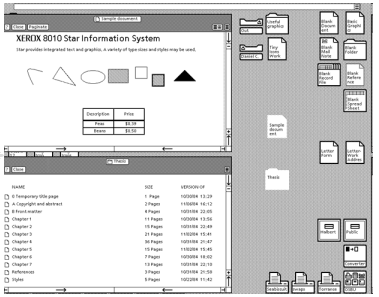
- Interpress, a resolution-independent graphical page-description language and the precursor to **PostScript**
- **Ethernet** as a local-area computer network
- Fully formed **object-oriented programming** (with class-based inheritance).
- **Model–view–controller** software architecture

And Xerox managers didn't understand them

Xerox PARC Researchers were very bright and open to share ideas

… History: Xerox PARC Inventions

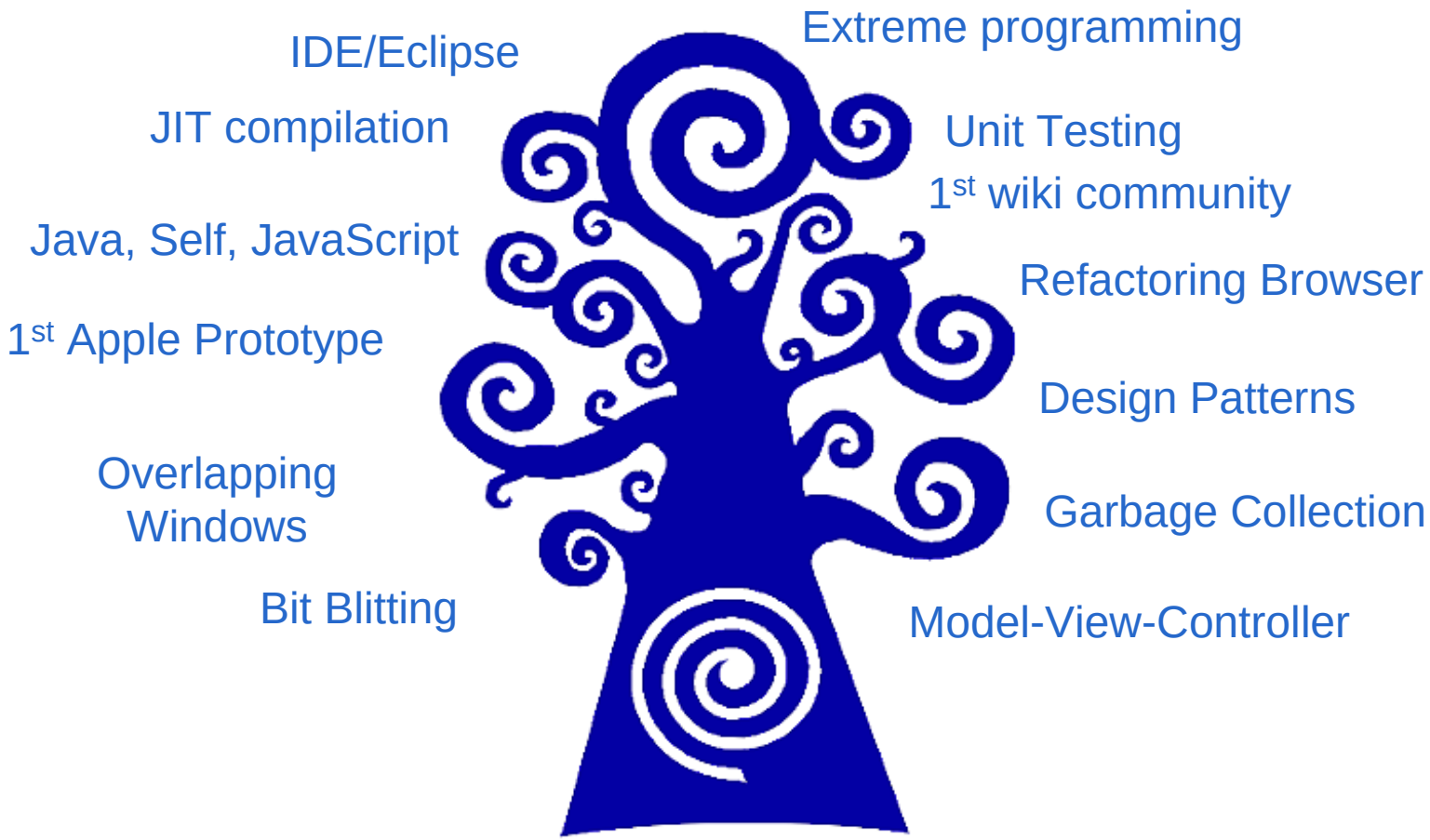
- Steve Jobs and Bill Gates have visited Xerox PARC
  - and both have been fascinated by GUI
- Jobs accused Microsoft later they have stolen the GUI concept from Apple



You have stolen the GUI concept from us

Well, Steve, I think there's more than one way of looking at it. I think it's more like we both had this rich neighbor named Xerox and I broke into his house to steal the TV set and found out that you had already stolen it.

… History: Xerox PARC Inventions



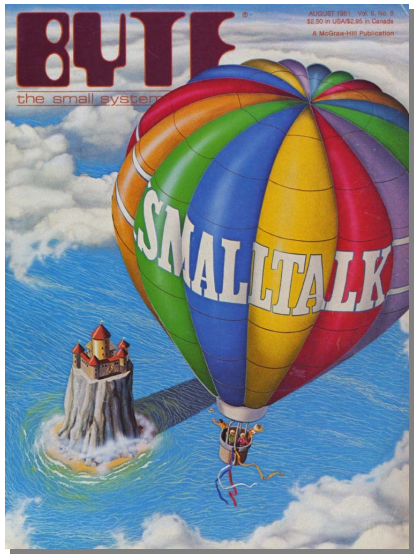
Objects, Garbage Collection, Byte Code, etc…

… History: And some name dropping

- Alan Kay
  - Smalltalk Inventor, Vision of a portable computer
- Dan Ingalls
  - Smalltalk Inventor
- Adele Goldberg
  - Smalltalk Inventor and writer of the 4 books
- Kent Beck
  - Founder of Extreme and Agile Development Initiative
- Ward Cunningham
  - Wiki ("The Wiki Way") and Agile Development Initiative
- Erich Gamma
  - Design Patterns & Eclipse
- Martin Fowler
  - Design Pattern & Software Development Methodology

… History: August 1981 Byte Magazine

- Byte Magazine issue in August 1981
  - Completely devoted to the Smalltalk-80
  - Article was written by Dan Ingalls and entitled "Design Principles Behind Smalltalk".
  - It provides an overview of the core ideas of Smalltalk.
- This article is still considered required reading for anyone new to Smalltalk.
- The cover of Byte Magazine shows a multi-colored balloon leaving the ivory tower of Xerox labs.
- Byte Magazine:
  - Most important computer Magazine at that time
  - Monthly edition up to 420'000 copies
  - Up to 1000 pages thick



https://archive.org/details/byte-magazine-1981-08

… History: The History outside Xerox PARC

- **1980 — Smalltalk-80**
  - ASCII, cleaning primitives for portability, metaclasses, blocks as first-class objects, MVC.
  - Projects: Gallery Editor (mixing text, painting and animations) + Alternate Reality Kit (physics simulation)
- **1981 — Books + 4 external virtual machines**
  - Dec, Apple, HP and Tektronix
  - GC by generation scavenging
- **1988 — Creation of Parc Place Systems**
- **1992 — ANSI Draft**
- **1995 — New Smalltalk implementations**
  - Dolphin, **Squeak**, Smalltalk/X, GNU Smalltalk
- **2002 — Seaside Smalltalk based Web Framework**
- **2008 — Pharo fork of Squeak**
  - clean up libraries
  - apply MIT Free software license

Smalltalk Environment

What is the "Smalltalk Environment"

School of Engineering

Smalltalk is a consistent, uniform world, written in itself

- All the source code is there all the time
- You can't lose code
  - except when environment crashes
- You can change everything

Everything is an object or a message

Everything happens by sending messages to objects

… Everything is an Object or a Message

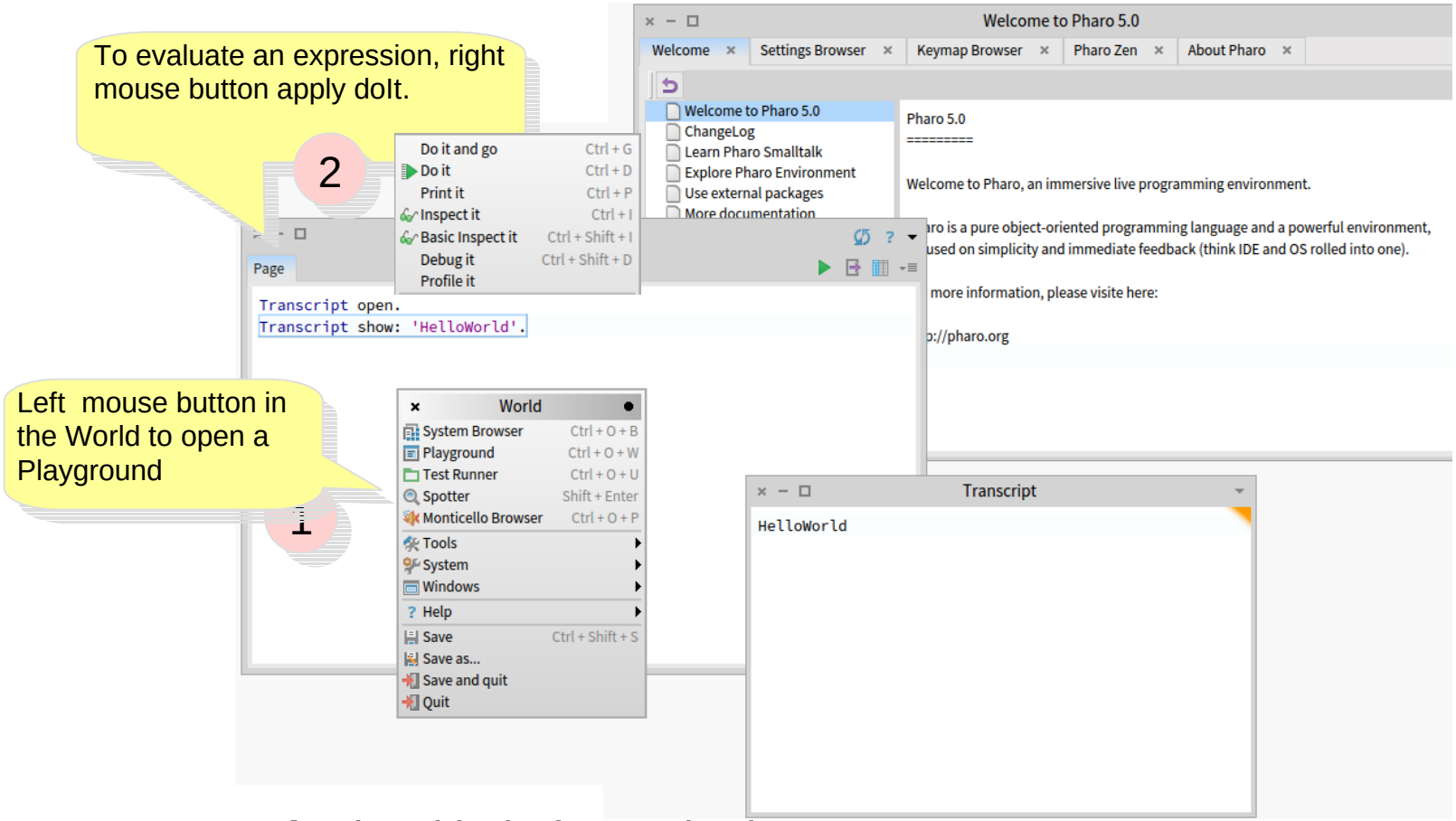
School of Engineering

- The workspace is an object.
- The window is an instance of SystemWindow.
- The text editor is an instance of ParagraphEditor.
- `'hello word'` is an instance of String.
- The mouse is an object.
- The parser is an instance of Parser.
- The compiler is an instance of Compiler.
- The process scheduler is also an object.

All the code is available, readable and *changeable at runtime*.

Hello World

- We can dynamically ask the system to evaluate an expression.

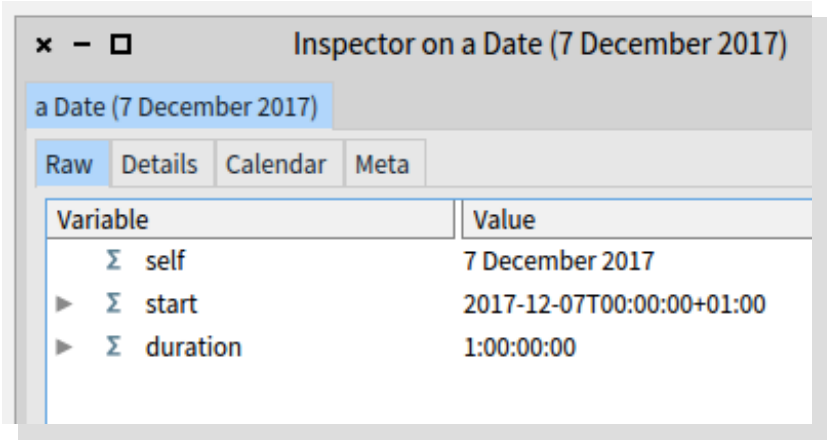


- **Transcript** is a kind of “standard output”

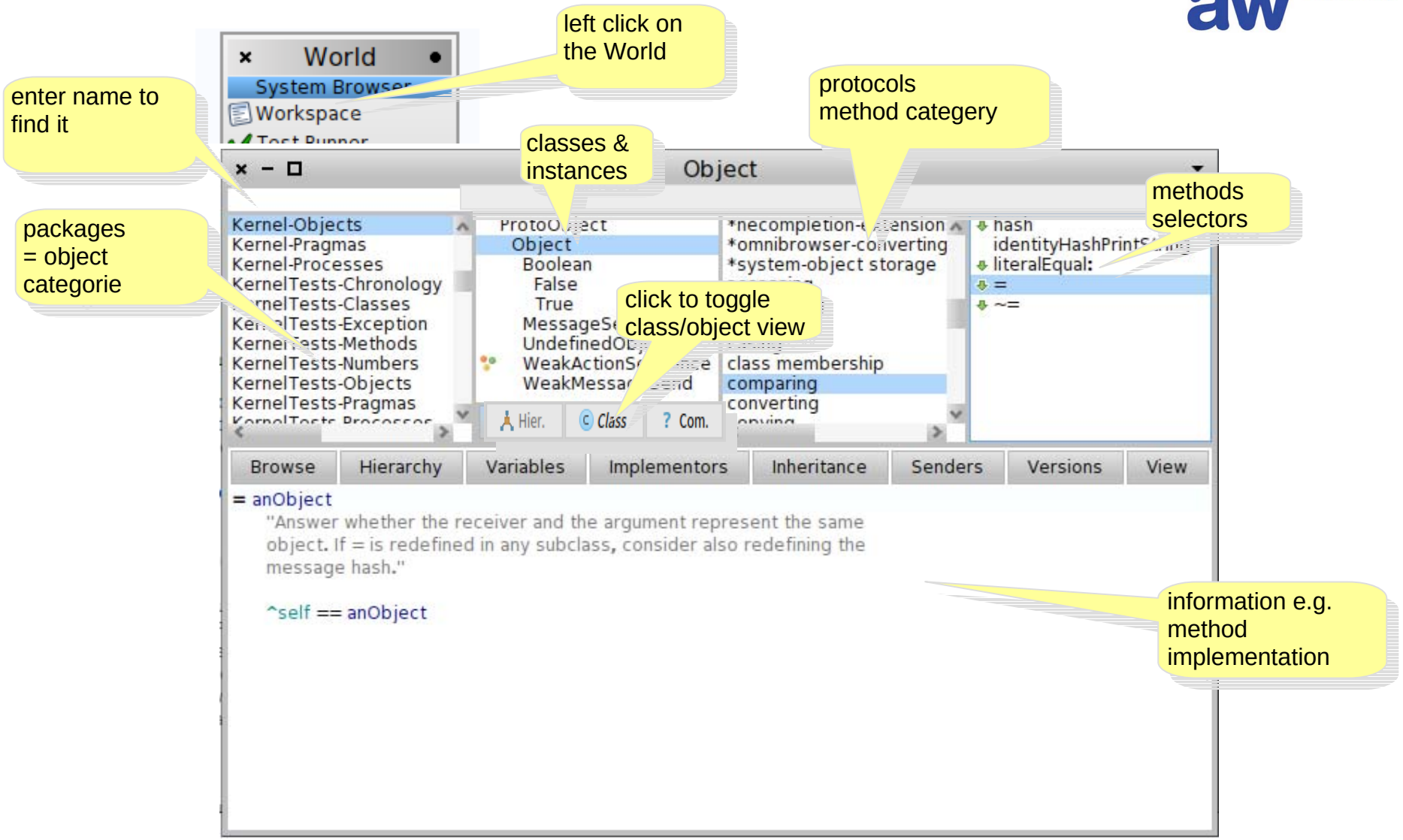
Environment

- Every object understands the message 'explore'. As a result, you get an Explorer window that shows details about the object."
- **`Date today explore.`**

- This shows that the date object consists of a point in time (start) and a duration (one day long).

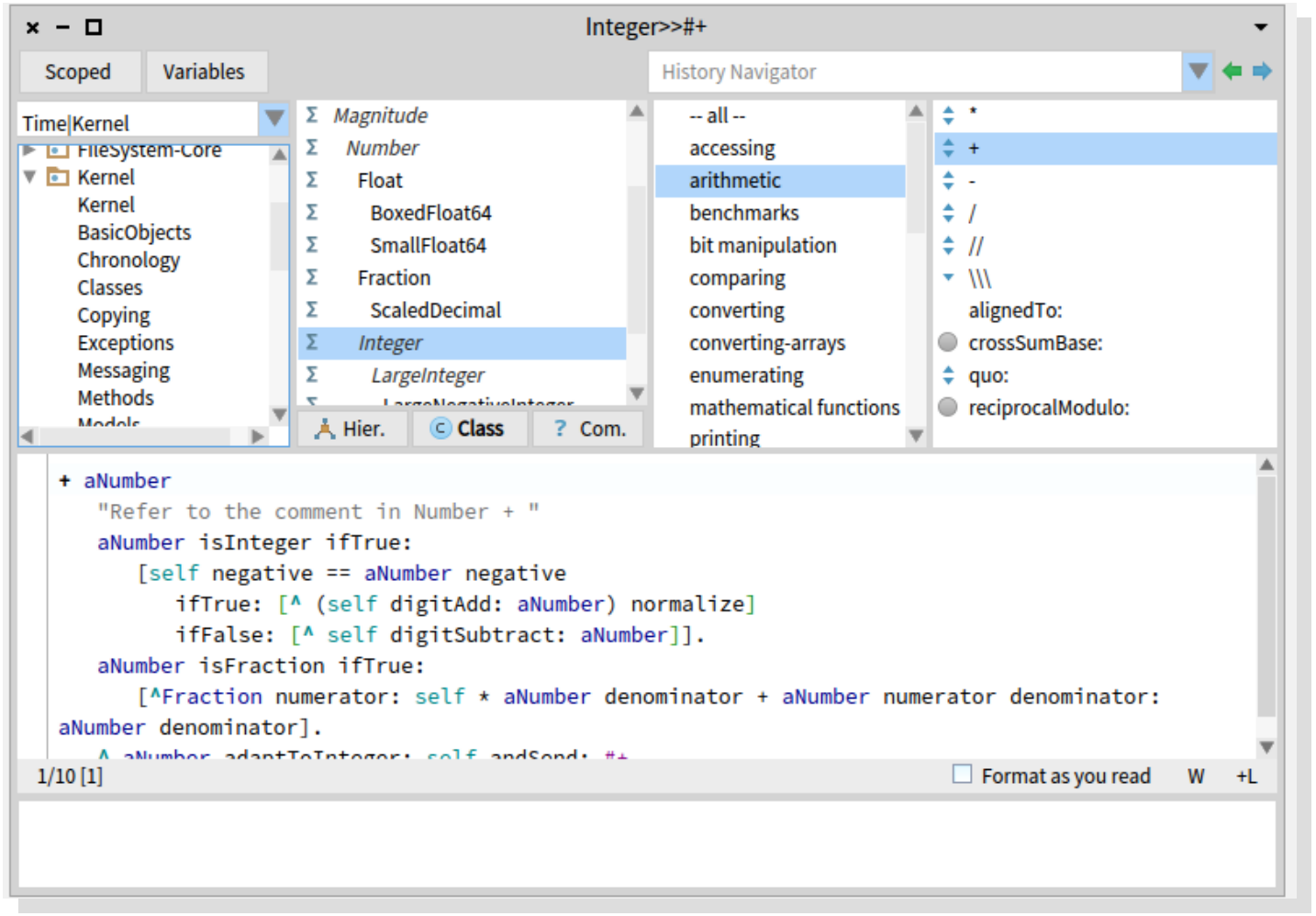


The Smalltalk System Browser & Editor



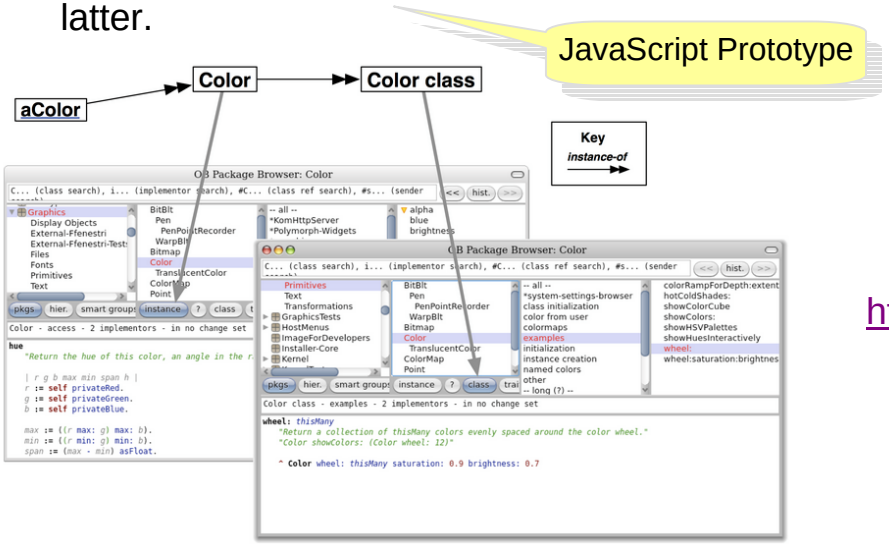
… the Smalltalk System Browser & Editor

- Every class/object in the Smalltalk system can be inspected and changed (!)



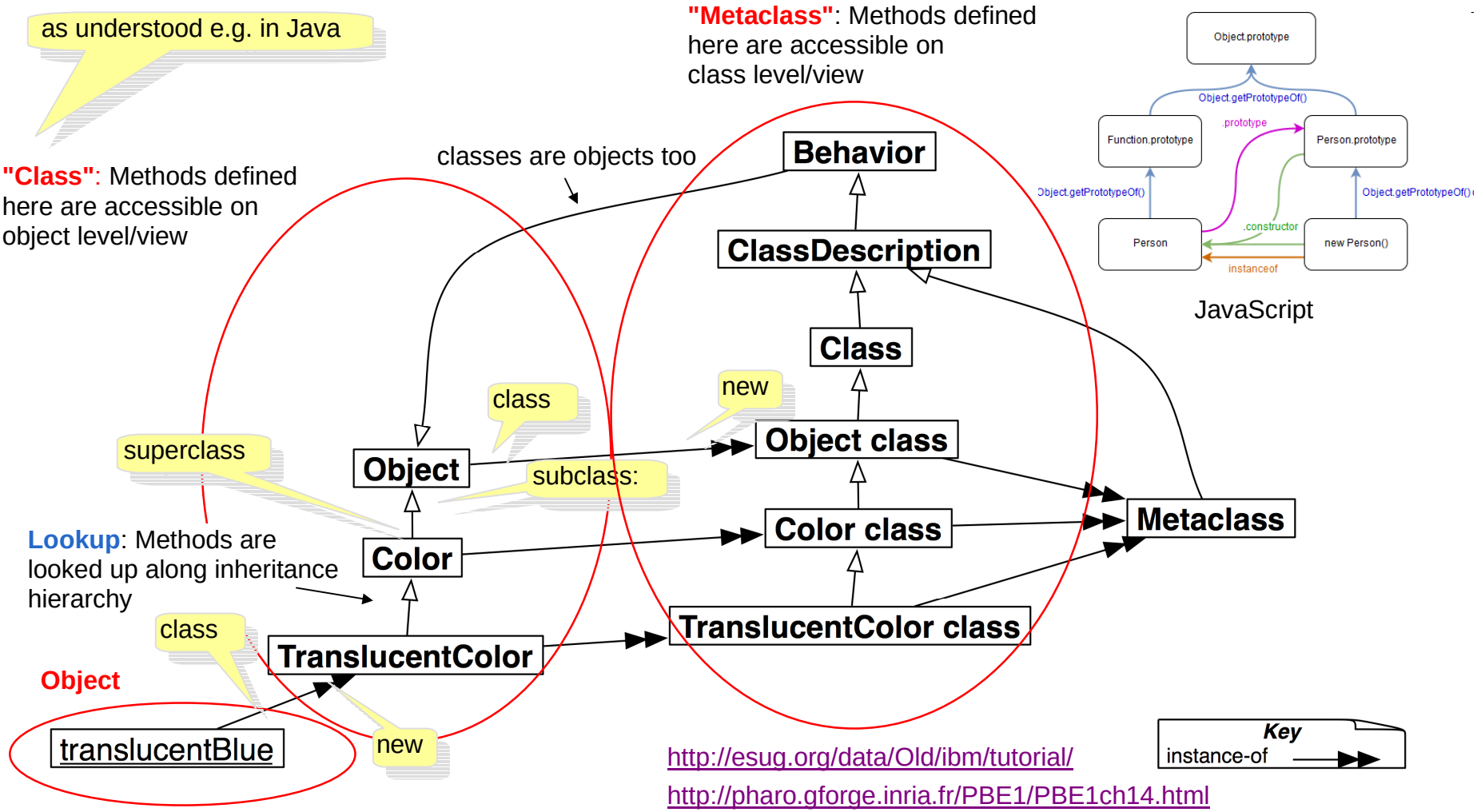
… the Smalltalk System Browser & Editor

- A class defines the structure of its instances i.e. object
  - i.e. instance variables and methods.
- Instance side and the class side
  - **Classes are objects too**, and every **object is an instance of a class**
  - Since classes are objects
    - *can have their own "instance variables" and their own methods.*
    - *these are called **class variables** and **class methods**,*
      - class variables/methods are just instance variables/methods defined by a **metaclass**
  - A ***class*** and its ***metaclass*** are two separate classes, even though the former is an instance of the latter.



http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html

The Smalltalk Object Model



http://esug.org/data/Old/ibm/tutorial/

http://pharo.gforge.inria.fr/PBE1/PBE1ch14.html

http://swing.fit.cvut.cz/projects/stx/doc/online/english/programming/doingThingsInST.html

Navigate and Construct at runtime

School of Engineering

- `c := Color new.`
- `c class.  Color`
- `Color class. Color class`
- `Color class class. Metaclass`
- `Color superclass. Object`

- `Color subclass: #TranslucentColor2.`
- `TranslucentColor2 superclass. Color`
- `TranslucentColor2 compile:'hello Transcript show: ''hello''.'.`
- `TranslucentColor2 compile:'hello2: arg Transcript show: arg.'.`

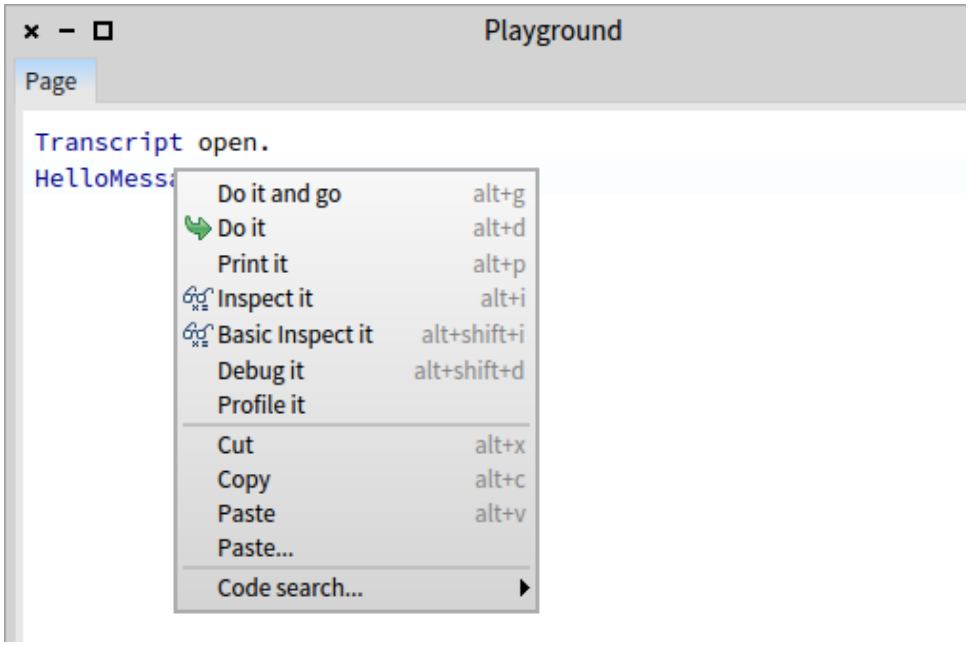
- `t := TranslucentColor2 new.`
- `t hello. hello`
- `t hello2: 'hugo'. hugo`

https://live.exept.de/doc/online/english/programming/doingThingsInST.html

Natural Language "Alike"

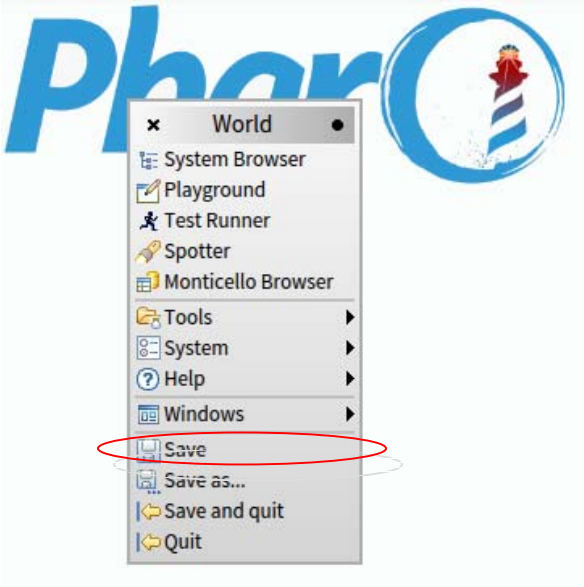
Accept, DoIt, PrintIt and InspectIt

- *There is a context menu for **selected strings***
- *DoIt*
  - Evaluate an expression
- *PrintIt*
  - Evaluate an expression and print the result
- *InspectIt*
  - Open in Object Explorer



Persistent Object Memory

- Smalltalk is language *and* environment
- Everything inside the image (workspace)
  - persistent objects
  - fully reflective system
  - incremental compilation



https://itnext.io/javascript-vs-pharo-d4fbf15578ee?gi=71920fe27043

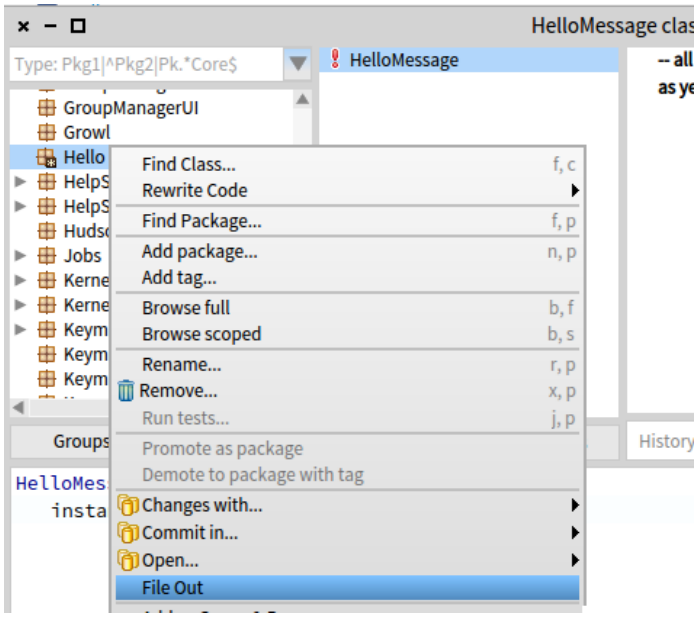
"JavaScript is a catastrophe of language design.

JavaScript is riddled with dark corners, traps and pitfalls. "

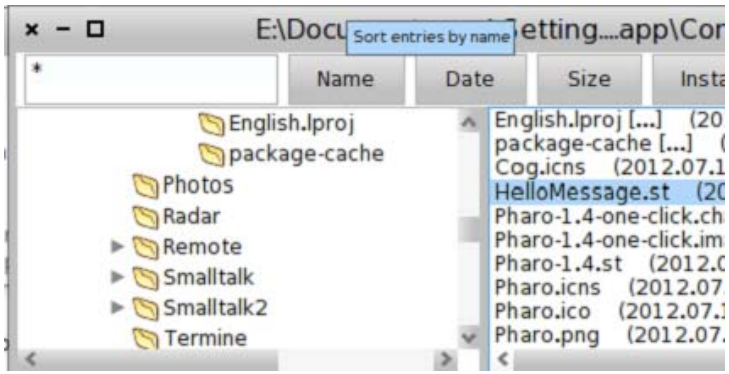
https://thenewstack.io/brendan-eich-on-creating-javascript-in-10-days-and-what-hed-do-differently-today/

Save & Load Single Classes

- To save in System Browser
- -> "File out"



- load a class: in World -> Tools -> FileBrowser "Install into ChangeSet"



The Language

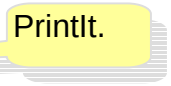
Objects in Smalltalk

School of Engineering

- ***Everything is an object***
  - Things only happen by message passing
  - Variables are dynamically typed
  - classes are also objects;
    - *the "class" is a metaobject that is constructed automatically when a class*
- State is private to objects
  - “protected” for subclasses
- (All) Methods are public
  - “private” methods by convention only
- (Nearly) every object is a reference
  - Unused objects are garbage-collected
- Single inheritance

Doing vs Printing

- DoIt
  - execute the expression
- PrintIt It's value
  - Do It which prints the result next to the expression you've selected.(`#printOn:`)



- `1 + 2.` `3`
- `Date today.` `13 March 2013`
- `Time now.` `11:42:23.487 am`
- `SmalltalkImage current datedVersion.` `' Pharo1.4 of 18 April 2012'`

Numbers, Characters, Strings and Symbols

Basic Types: Numbers

- 1, 2, 100, 2/3 ... are Numbers, and respond to many messages evaluating mathematical expressions. Evaluate these ones:

Examples

```
2. 2
20 factorial.  2432902008176640000
1000 factorial / 999 factorial. 1000
(1/3).  (1/3)
(1/3) + (4/5).  (17/15)
(1/3) asFloat.  0.3333333333333333
1 class maxVal.  1073741823
```

Try this in Java!

School of Engineering

1000 factorial

```
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404
800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779
505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012
476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207
379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281
231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261
683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909
342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945
160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034
352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223
838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779
911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210
465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327
168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860
788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084
024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346
962715422845862377387538230483865688976461927383814900140767310446640259899490222221765904339901886
018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646124960
798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281
434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506
217095002597389863554277196742822248757586765752344220207573630569498825087968928162753848863396909
959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941872748998
094254742173582401063677404595741785160829230135358081840096996372524230560855903700624271243416909
004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Automatic Coercion

School of Engineering

- On demand numeric types are coerced automatically

```
1 + 2.3                                  3.3
1 class                                  SmallInteger
1 class maxVal class                     SmallInteger
(1 class maxVal + 1) class               LargePositiveInteger
(1/3) + (2/3)                            1
1000 factorial / 999 factorial           1000
2/3 + 1                                  (5/3)
(1/3) class.                             Fraction
```

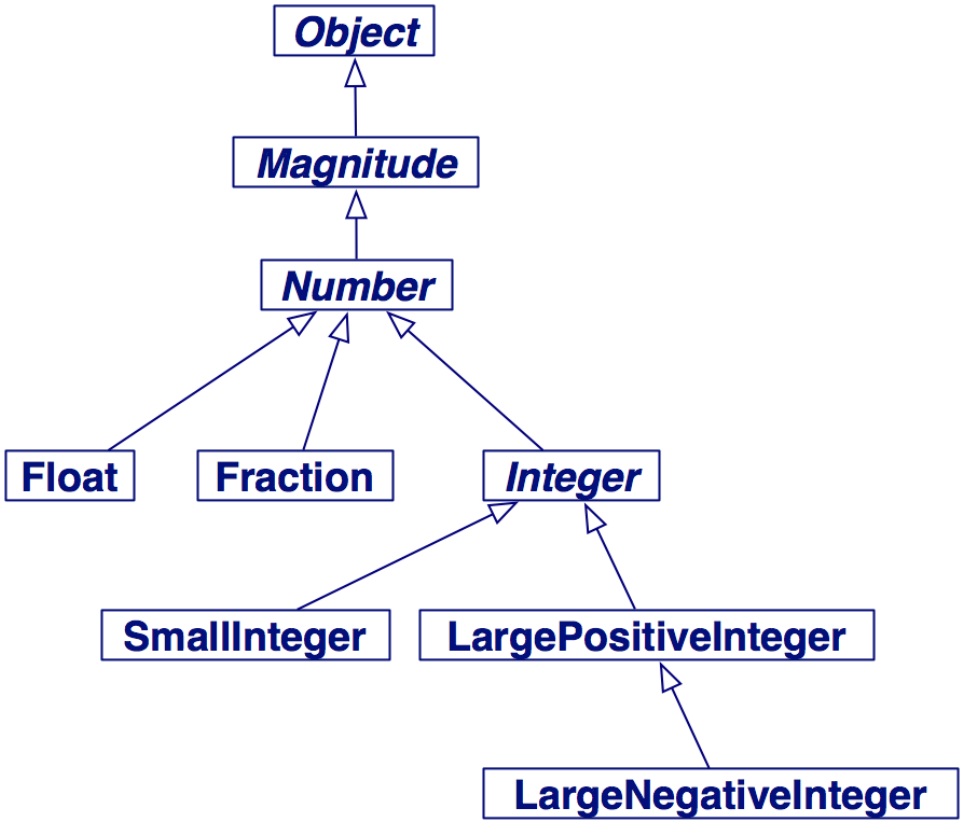
Explicit Coercion

- Types can also be converted explicitly

```
d := i asFloat.
i := d asInteger.
i := d truncated.
i := d rounded.
s := i asString.
```

Numbers

- Object hierarchie of numerical types



Basic Types: Characters

- A Character can be instantiated using $ operator:

Examples

```
$A. $A
$A class. Character
$B charCode. 66
print all 256 characters of the ASCII extended set:"
Character allByteCharacters.
   '!"#$%&''()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
   abcdefghijklmnopqrstuvwxyz{|}~□?□??????????□?□□????????????□?? ¡¢
   £¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãä
   åæçèéêëìíîïðñòóôõö÷øùúûüýþÿ'
```

Unprintable Characters

- Unprintable characters:

```
Character space
Character tab
Character cr
10 asCharacter
```

Basic Types Strings

- A String is a collection of characters.
- Single quotes to create a String object.

Examples

```
'ProfStef'.  'ProfStef'
'ProfStef' size.  8
'abc' asUppercase. 'ABC'
'Hello World' reverse.  'dlroW olleH'
```

You can access each character using at: message

```
'ProfStef' at: 1.  $P
```

String concatenation uses the comma operator:

```
'ProfStef', ' is cool'.  'ProfStef is cool'
```

String concatenation.

Strings

Strings are mutable

```
s := 'Hello World'.
s at: 4 put: $L.                    'HelLo World'

12 printString                      '12'
String with: $A                     'A'
'can''t' at: 4                      $'
'hello', ' ', 'world'               'hello world'
```

Basic Types Symbols, Comparison

School of Engineering

- A Symbol is a String which is guaranteed to be globally unique.
- There is one and only one Symbol #ProfStef. There may be several 'ProfStef' String objects.
- Message == returns true if the two strings are the SAME OBJECT
- Message = returns true if the strings are EQUAL

Examples

```
'ProfStef' asSymbol.  #ProfStef
#ProfStef asString.  'ProfStef'
(2 asString) == (2 asString). false
(2 asString) asSymbol == (2 asString) asSymbol.  true
```

Symbols vs. Strings

School of Engineering

- Symbols are used as method selectors and unique keys for dictionaries
  - Symbols are read-only objects, strings are mutable
  - A symbol is unique, strings are not
  - Symbol operations are faster than String operations
  - Conversion of a String to a Symbol is expensive

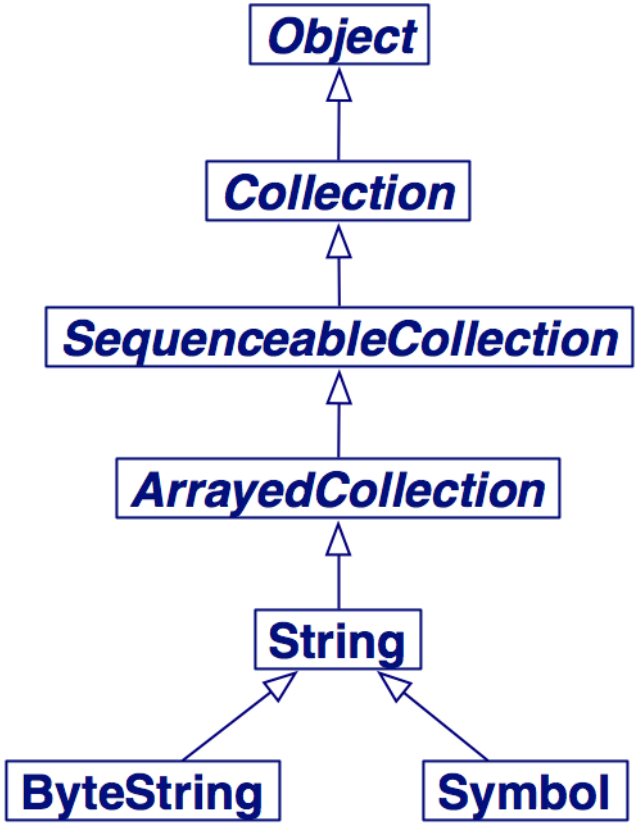
```
'calvin' = 'calvin'                    true
'calvin' == 'calvin'                   true  !
'cal','vin' = 'calvin'                 true
'cal','vin' == 'calvin'                false

#calvin = #calvin                      true
#calvin == #calvin                     true
#cal,#vin = #calvin                    true
#cal,#vin == #calvin                   false
#cal,#vin                              'calvin'
(#cal,#vin) asSymbol == #calvin        true
```

Identity vs. Equality

- = tests Object value
  - Should normally be overridden
    - *Default implementation is == !*
  - You should override hash too!
- == tests Object identity
  - *Should never be overridden*

Strings


Object
Collection
SequenceableCollection
ArrayedCollection
String
ByteString
Symbol

Comments

- Comments are simply enclosured in ""

Examples

- **`"This is a Smalltalk comment"`**

Arrays

Basic Type: Array

- Literal arrays are created at parse time

Examples

```
#(1 2 3).  #(1 2 3)
#( 1 2 3 #(4 5 6)) size. 4
#(1 2 4) isEmpty. false
#(1 2 3) first. 1
```

- Arrays are mutable

```
#('hello' 'Squeak') at: 2 put: 'Pharo'; yourself.  #('hello' 'Pharo')
```

- Array Index start at 1 !!!

Dynamic Arrays

- create array with up to 4 elements

```
x := Array with: 5 with: 4 with: 3 with: 2.
```

- allocate an array with specified size

```
x := Array new: 4.
```

set array elements

```
x
   at: 1 put: 5;
   at: 2 put: 4;
   at: 3 put: 3;
   at: 4 put: 2.
```

Boolean

True

School of Engineering

- True (and False) are special classes in Smalltalk
- The result of a comparison is an Object of this kind
- `(3 > 2) class. True`
- They understand ifTrue and ifFalse messages
  - The following Block is compiled inline and executed
- `(3 > 2) ifTrue: [Transcript  show: 'hello']`
- not and & without lazy evaluations

```
True>>ifTrue: trueBlock ifFalse: falseBlock
   "Answer with the value of trueBlock.
   Execution does not actually reach here
   because the expression is compiled in-line."

   ^ trueBlock value
```

true and false

- **`true`** and **`false`** are unique instances (singletons) of `True` and `False`
  - Optimized and inlined

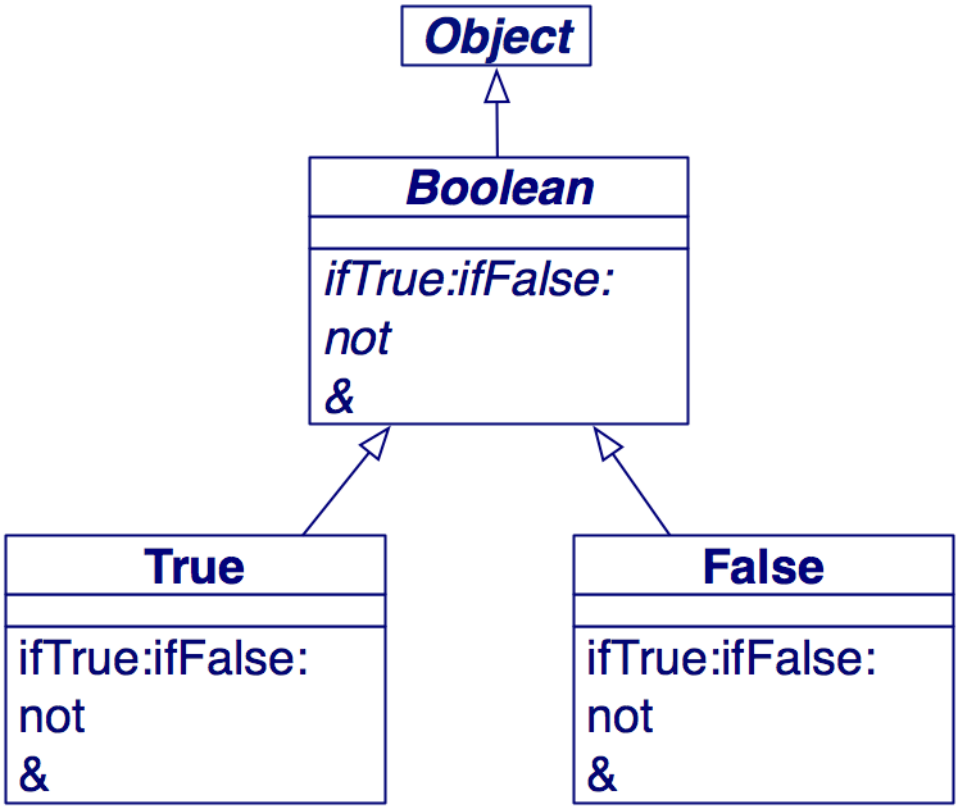
- Lazy evaluation with `and:` and `or:`

```
false and: [1/0]
```
`false`

Block that is evaluated lazy

```
false & (1/0)
```
`ZeroDivide error!`

Booleans


Object
Boolean
ifTrue:ifFalse:
not
&
True
ifTrue:ifFalse:
not
&
False
ifTrue:ifFalse:
not
&

Variables

Local Variables

- Declare local variables with **| … |**
- only name required

```
| x y |
```

- Use := to assign a value to a variable

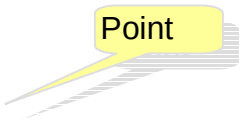
```
x := 1
```

- Old fashioned assignment operator (in original Books): ←

Assignment

School of Engineering

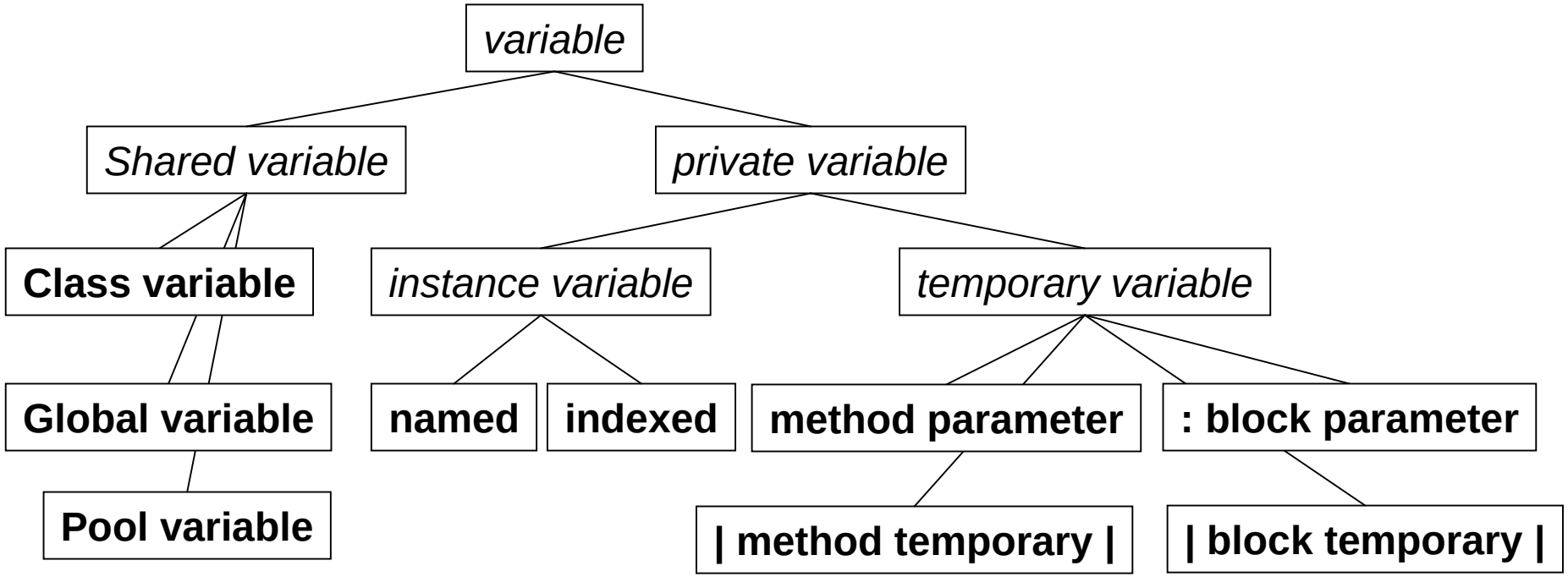
- Assignment binds a name to an object reference
- Method arguments cannot be assigned to!
  - Use a temporary instead
- Different names can point to the same object!
  - Assignment only copies references
  - Watch out for unintended side effects




```
|p1 p2|
p1 := 3@4.
p2 := p1.
p1 setX: 5 setY: 6.
p2                                  5@6
```

Variables

- A variable maintains a reference to an object
  - Dynamically typed
  - Can reference different types of objects
  - Shared (initial uppercase) or local (initial lowercase)



Pseudo-Variables

- The following pseudo-variables are hard-wired into the Smalltalk compiler.

`nil`	A reference to the UndefinedObject
`true`	Singleton instance of the class True
`false`	Singleton instance of the class False
`self`	Reference to this object Method lookup starts from object's class
`super`	Reference to this object (!) Method lookup starts from the superclass

Messages

Message Syntax: Unary Messages

School of Engineering

- Messages are sent to objects.
- Are also called Selectors
- There are three types of message: Unary, Binary and Keyword.

- Unary messages have the following form: *anObject aMessage*

Examples

```
1 class.  SmallInteger
false not.  true
Time now.  1:05:49.375 pm
Date today.  13 March 2013
Float pi.  3.141592653589793
```

Message Syntax: Binary Messages

- Binary messages have the form: anObject + anotherObject

Examples

```
3 * 2.  6
Date today + 3 weeks.  3 April 2013
false | false.  false
true & true.  true
true & false.  false
10 @ 100.  (10@100)
10 <= 12.  true
'ab', 'cd'.  'abcd'
Date today < Date yesterday.  false
```

Message Syntax: Keyword Messages

- Keyword Messages are messages with arguments
- They have the following form:
  anObject akey: anotherObject [akey2: anotherObject2]

Examples

```
1 max: 3.  3
4 between: 0 and: 10. true
Point new setX:4 setY:5. (4@5)
Color r:1 g:0 b:0.  Color red
Color
   r:1
   g:1
   b:0.  Color yellow
```

The message is between:and: sent to the Number 4

"The message is r:g:b: implemented on class Color.

Message Syntax: Execution Order

School of Engineering

- Unary messages are executed first, then binary messages and finally keyword messages:

Unary > Binary > Keywords

```
2 + 3 squared.  11
2 raisedTo: 3 + 2.  32 !!
(0@0) class.  Point
0@0 corner: 100@200.  (0@0) corner: (100@200)
(0@0 corner: 100@200) class.  Rectangle
```

- Messages of similar precedence, expressions are executed from left to right

```
-3 abs negated reciprocal.  (-1/3)
```

Message Syntax: Parentheses

- Parentheses are used to change order of evaluation

```
(2 + 3) squared. 25
(2 raisedTo: 3) + 2. 10
(0@0 extent: 100@200) bottomRight. (100@200)
```

Mathematical Precedence

School of Engineering

- Traditional precedence rules from mathematics do not follow in Smalltalk.

```
2 * 10 + 2. 22
```

- The message * is sent to 2, which answers 20, then 20 receive the message +

- All messages always follow a simple left-to-right precedence rule, without exceptions !.

```
2 + 2 * 10. 40
2 + (2 * 10). 22
8 - 5 / 2. (3/2)
(8 - 5) / 2. (3/2)
8 - (5 / 2). (11/2)
```

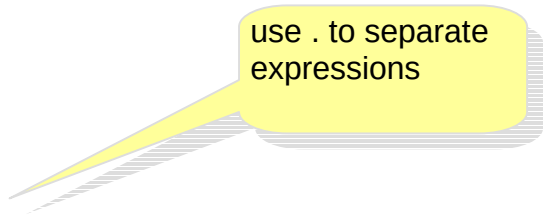
Message Call Syntax: Cascade

- ; is the cascade operator. It's useful to send message to the SAME receiver

- Open a Transcript (console):

```
Transcript open.

Transcript show: 'hello'.
Transcript show: 'Smalltalk'.
Transcript cr.
```



- is equivalent to:

```
Transcript
        show: 'hello';
        show: 'Smalltalk';
        cr.
```

Message Call Syntax: Cascade variants

School of Engineering

- cascade message to the result of the message

```
1 class maxVal.  1073741823
```

- Use *periods* to separate expressions

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr     "NB: don't need one here"
```

- Use *semi-colons* to send a cascade of messages to the same object

```
Transcript cr; show: 'hello world'; cr
```

Blocks

Blocks

- Blocks are **anonymous methods** that can be stored into variables and executed on demand.
- Blocks are delimited by square brackets: **[]**

Examples

executing block
nothing happens

```
[Transcript show: 'Hello'].
```

- does nothing because block is not executed.

```
[Transcript show: 'Hello'] value
```

- Blocks can have parameters :x
- A block that adds 2 to its argument (its argument is named x):

```
[:x | x+2].
[:x | x+2] value: 5. 7
[:x | x+2] value: 10.  12
[:x :y| x + y] value:3 value:5. 8.
```

Block Assignation

School of Engineering

- Blocks can be assigned to a variable then executed later.

- Note that |b| is the declaration of a variable named 'b' and that ':=' assigns a value to a variable.

```
|b|
b := [:x | x+2].
b value: 12.
```

```
14
```

Conditionals and Loops

Conditionals

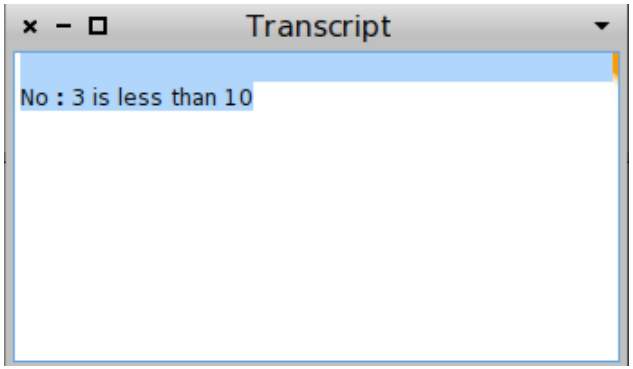
- Conditionals are messages sent to Boolean objects

Examples

```
1 < 2
   ifTrue: [100]
   ifFalse: [42].   100

3 > 10
   ifTrue: [Transcript show: 'maybe there''s a bug ....']
   ifFalse: [Transcript show: 'No : 3 is less than 10'].
```

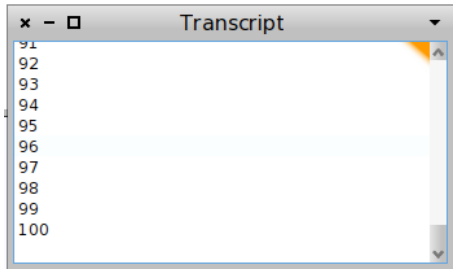
inserts ' in String



For Loops

- Loops are high-level collection iterators, implemented as regular methods.
- Basic loops:
- to:do:
- to:by:do:

Examples




```
1 to: 100 do:
  [:i | Transcript show: i asString; cr ].

1 to: 100 by: 3 do: [:i | Transcript show: i asString; cr].

100 to: 0 by: -2 do:
    [:i | Transcript show: i asString; cr].
```

While Loop

- Conditional expression must be in brackets here (unlike “if”)
- for the reason that it must be re-evaluated each time around the loop.

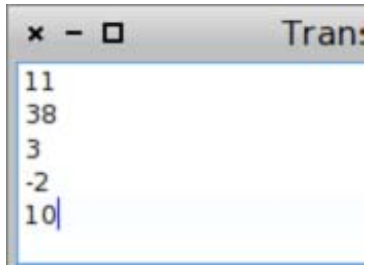
Example

```
[i < 100] whileTrue: [
sum := sum + i.
i := i + 1
]
```

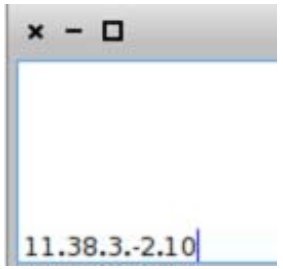
Iterators

- The message do: is sent to a collection of objects (Array, Set, OrderedCollection), evaluating the block for each element.

```
#(11 38 3 -2 10) do: [:each |
     Transcript show: each printString; cr].
```




```
#(11 38 3 -2 10) collect: [:each | each abs].  #(11 38 3 2 10)
#(11 38 3 -2 10) collect: [:each | each odd].  #(true false true
  false false)
#(11 38 3 -2 10) select: [:each | each odd].  #(11 3)
#(11 38 3 -2 10) select: [:each | each > 10].  #(11 38)
#(11 38 3 -2 10) reject: [:each | each > 10].  #(3 -2 10)
#(11 38 3 -2 10) do: [:each | Transcript show: each printString]
     separatedBy: [Transcript show: '.'].
```



Objects and Classes

Instantiation

```
SimpleButtonMorph new
```

- creates (and returns) a new instance of the MessagePublisher class.
- This is typically assigned to a variable:

```
button := SimpleButtonMorph new
```

- However, it is also possible to send a message to a temporary, anonymous object:

```
SimpleButtonMorph new openCenteredInWorld.
```

Instantiation

- The message #allInstances sent to a class delivers an Array with all instances of this class.
- **`SimpleButtonMorph allInstances size.`**

How many instances of SimpleButtonMorph exist

- After new the initial values can be set and methods can be invoked
-
```
SimpleButtonMorph new
    label: 'Open Transcript';
    target: [Transcript open.];
    actionSelector: #value;
    openCenteredInWorld.
```

create a new instance of a Button

command to be executed

and open it

- Change its label

```
SimpleButtonMorph allInstances last label: 'hello'
```

- Delete the Button

```
SimpleButtonMorph allInstances last delete.
```

Define new Classes

Send a #subclass Message to Class that should be inherited from

```
Object subclass: #HelloMessage
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Hello'!
```

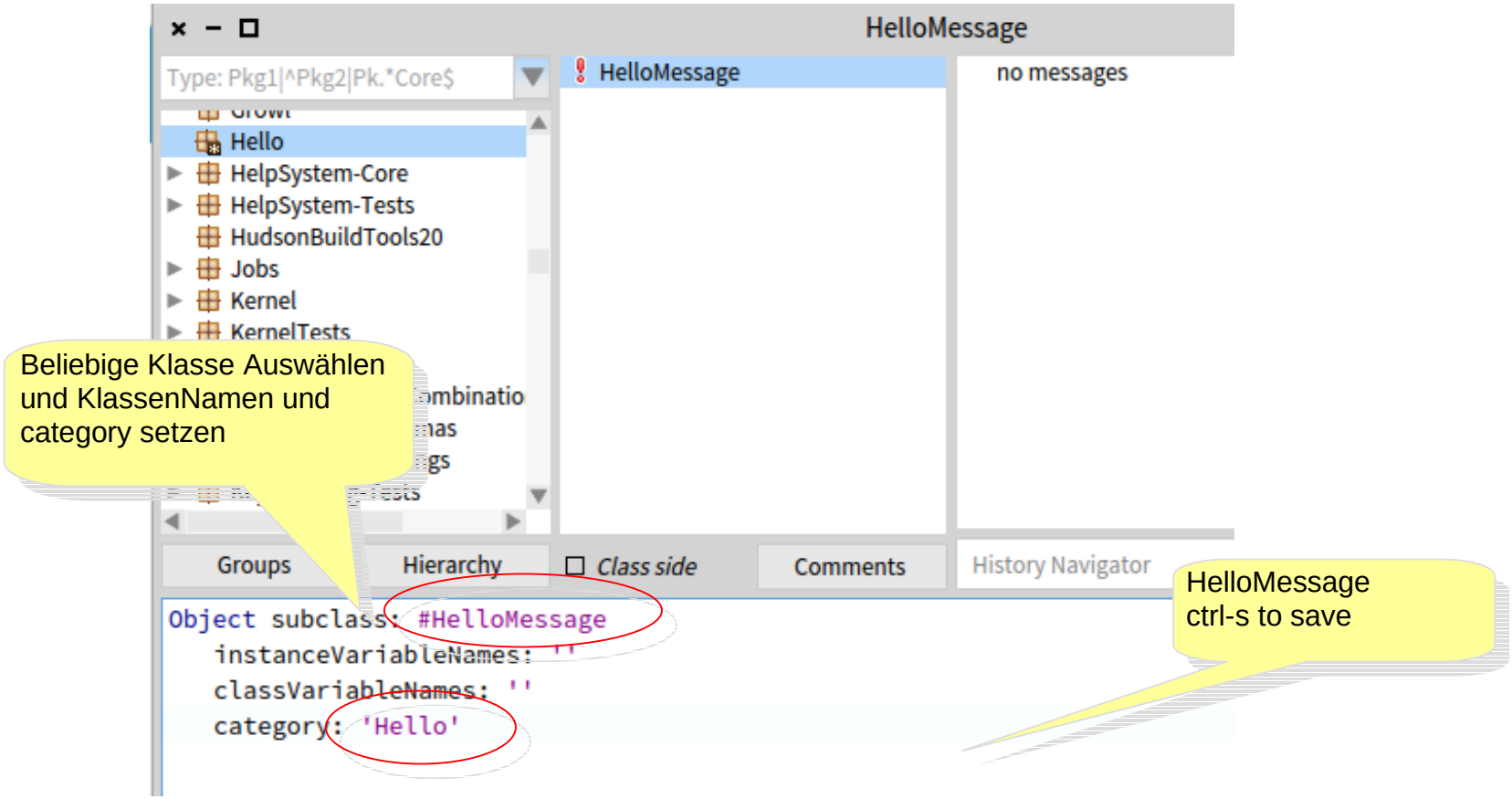
name of the new class

category of the class

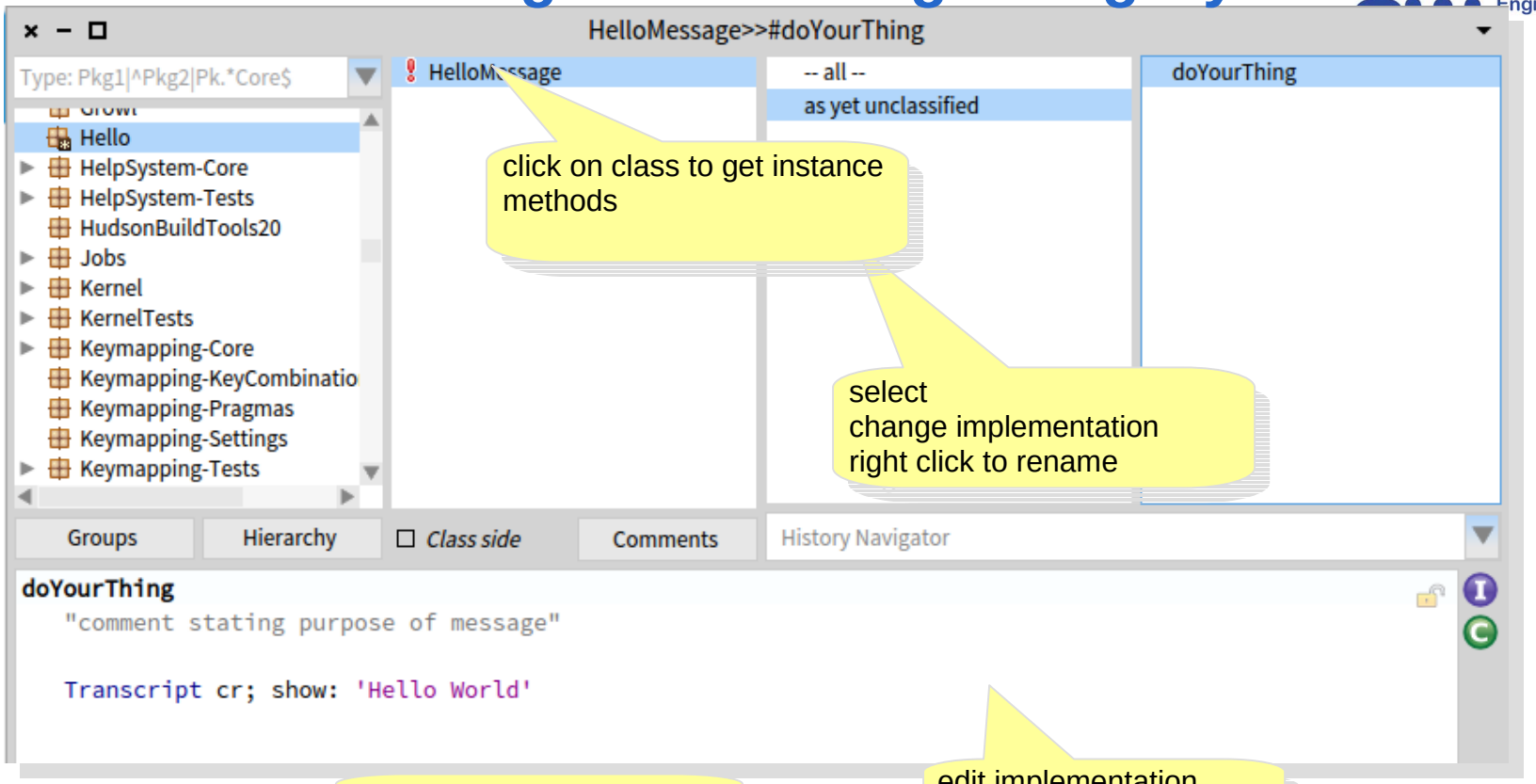
```
Number subclass: #Complex
    instanceVariableNames: 'real img'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ComplexNumbers'
```

names of the instance variables

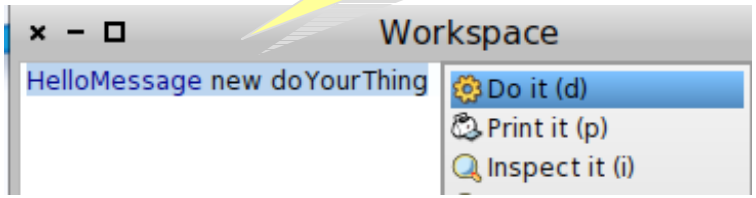
Define new Classes via System Browser



Add Instance Message in new Msg Category



- Excecute it



Instance Variables

- Are declared in the class definition (no type)

```
instanceVariableNames: 'real img'
```

- Instance variables are private to the instance itself
- Instance variables can be accessed by name
  - In any of the instance methods of the class that defines them,
  - In the methods defined in its subclasses.
    - *There is no language syntax that provides direct access to the instance variables of any other object.*

- To access them "accessor methods" have to be defined.

```
Complex >> real
^real
Complex >> real: val
real:=val
```

^ = "return"

>> indicates real is method of Complex; not part of Smalltalk Syntax but used in external text formats

Message and Variables

- *Local variables* within methods (or blocks) are delimited by |*var*|

```
OrderedCollection  >> collect: aBlock
    "Evaluate aBlock with each of my elements as the argument."
    | newCollection |
    newCollection := self species new: self size.
    firstIndex to: lastIndex do:
        [ :index |
        newCollection addLast: (aBlock value: (array at: index))].
    ^ newCollection
```

- *Block parameters* are delimited by **:*var*|**

```
[:n | x := n+1. y := n-1. x * y] value: 10          99
```

x and y are instance or method scoped variables

Return Value of Messages

- Use a *caret* (^) to return a value from a method or a block

```
max: aNumber
    ^ self < aNumber
        ifTrue: [aNumber]
        ifFalse: [self]
```

```
1 max: 2
```

2

- Methods *always* return a value
  - By default, methods return `self`

Return Value of Messages Examples

School of Engineering

- Example

```
Point >> dist: aPoint
    "Answer the distance between aPoint and the receiver."
    | dx dy |
    dx := aPoint x - x.
    dy := aPoint y - y.
    ^ ((dx * dx) + (dy * dy)) sqrt

Complex >> + aComplex
   | nComplex |
   nComplex := Complex new.
   nComplex real: (self real + aComplex real).
   nComplex img: (self img + aComplex img).
   ^ nComplex
```

Object Instance Methods

School of Engineering

- Every Object supports following messages (and more)

`class`	returns the receiver class
`isKindOf: aClass`	whether aClass is a superclass of the receiver
`respondsTo: aSymbol`	whether the class or its superclasses understands the message
`==`	comparison of two objects
`=`	comparison of two object values
`isNil`	test if object is nil
`copy`	copy of an Object
`shallowCopy`	a shallow copy of the object
`deepCopy`	a deep copy of the object

Answering on Print Message

- Answering PrintOn Message
- Send #PrintString to an object to convert it to string using PrintOn

```
Complex >> printOn: aStream

   aStream nextPutAll: 'real:'.
   real printOn: aStream.

   aStream nextPutAll: ' img:'.
   img printOn: aStream.
```

Examples

Slow Fibonacci

- Recursive Method

```
Fibs >> at: anIndex
   anIndex = 1 ifTrue: [ ^ 1 ].
   anIndex = 2 ifTrue: [ ^ 1 ].
   ^ (self at: anIndex - 1) + (self at: anIndex - 2)
```

```
Fibs new at: 35   9227465
```

Takes 8 seconds.
Forget about larger values!

Caching Fibonacci

- A Dictionary to cache values

```
Object subclass: #Fibs
   instanceVariableNames: 'fibCache'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Misc'

Fibs >> initialize
   fibCache := Dictionary new

Fibs >> fibCache
   ^ fibCache
```

called automatically when new instance created

Caching Fibonacci

- Now we introduce the lookup method, and redirect all accesses to use the cache lookup

```
Fibs >> lookup: anIndex
   ^ self fibCache at: anIndex ifAbsentPut: [ self at: anIndex ]

Fibs >> at: anIndex
   anIndex = 1 ifTrue: [ ^ 1 ].
   anIndex = 2 ifTrue: [ ^ 1 ].
   ^ (self lookup: anIndex - 1) + (self lookup: anIndex - 2)
```

```
Fibs new at: 100   354224848179261915075
```

… is virtually instantaneous!

Conclusion

School of Engineering

- Smalltalk is a unique *Language* and *Environment*
- The Language
  - "Purely" Object Oriented
  - Based on a small set of concepts
  - Has influenced most modern OO programming languages (Java, C#)
  - Basis of Scratch
- The System
  - Concept of a open and arbitrary modifiable Workspace
    - *Everything can be inspected*
    - *Everything can be changed*
- Despite its age there is an actual and living community that maintains it

Questions

License

http://creativecommons.org/licenses/by-sa/3.0/

A Simple Drawing Canvas

```
Morph subclass: #CanvasMorph
    instanceVariableNames:
   'drawBlock'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ZHAW'
```

called automatically when new instance created

```
>> initialize
    super initialize.
    self color: Color white.
    self extent: 400@400.

>> drawBlock: aBlock
    drawBlock := aBlock.
    self changed.
```

"repaint"

"paint"

```
>> drawOn: aCanvas
    super drawOn: aCanvas.
    drawBlock ifNotNil: [
        aCanvas
            translateBy: self
bounds origin
            clippingTo: self bounds
            during: [:canvas |
drawBlock value: canvas]]
```

"Graphics"

```
use in Workspace
m := CanvasMorph new.
m openInWindowLabeled: 'Hallo'.
m drawBlock: [ .... ]
m delete.
```

open

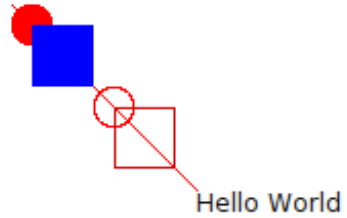
graphic passed as block

close

Drawing Samples

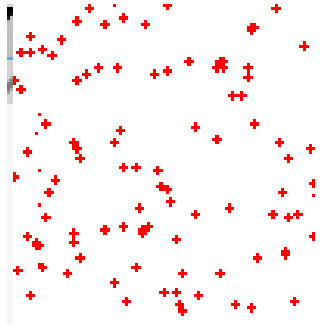
- some sample Canvas Drawing

```
m drawBlock: [:c |
   c line: 10@10 to: 100@100 color: Color red.
   c frameOval: (50@50 extent: 20@20) color:  Color red.
   c fillOval: (10@10 extent: 20@20) color:  Color red.
   c frameRectangle: (60@60 extent: 30@30) color:  (Color r:0.8 g:0 b:0).
   c fillRectangle: (20@20 extent: 30@30) color:  Color blue.
   c drawString: 'Hello World' at: 100@100.
]
```



- draw 100 Random Crosses

```
m drawBlock: [:c |
   rand := Random new.
   1 to: 100 do: [: i |
        x := (rand next * 100).
        y := (rand next * 100).
        c line: (x-1)@y to: (x+1)@y color: Color red.
        c line: x@(y-1) to: x@(y+1) color: Color red.
   ]
]
```



Stack Implemented as Array

```
Object subclass: #Stack
  instanceVariableNames:
    'anArray top '
  classVariableNames: ''
  poolDictionaries: ''  !


!Stack class methods !

new
        | s |
    s := super new.
    s setsize: 10.
    ^s !!
```

```
!Stack methods !

pop
        | item |
    item := anArray at: top.
    top := top - 1.
    ^item!

printOn: aStream
    aStream nextPutAll: 'Stack['.
    1 to: top do: [:i | (anArray
   at: i) printOn: aStream.
   aStream space].
    aStream nextPutAll: ']'!

push: item
    top := top + 1.
    anArray at: top put: item!

setsize: n
   anArray := Array new: n.
    top := 0
```