The Object-Oriented Programming Paradigm

Smalltalk

Bent Thomsen

Slides mainly based on material
by Prof. O. Nierstrasz, U. Bern

The Object-Oriented Programming Paradigm

- Characteristics:
  - Discipline and idea
    - *The theory of concepts, and models of human interaction with real world phenomena*
- Universal encapsulation construct
    - *Data structure*
    - *File system*
    - *Database*
    - *Window*
    - *Integer*
  - Everything is an object
- Metaphor usefully ambiguous
  - sequential or concurrent computation
  - distributed, sync. or async. communication

The Object-Oriented Programming Paradigm

- Characteristics:
  - Data as well as operations are *encapsulated* in objects
  - *Information hiding* is used to protect internal properties of an object
  - Objects interact by means of *message passing*
    - *A metaphor for applying an operation on an object*
  - In most object-oriented languages objects are grouped in classes
    - *Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects*
    - *Classes represent concepts whereas objects represent phenomena*
  - Classes are organized in *inheritance* hierarchies
    - *Provides for class extension or specialization*

4 major OOP principles

> Data Abstraction
> Encapsulation
> Inheritance
> Polymorphism
  — Dynamic dispatch

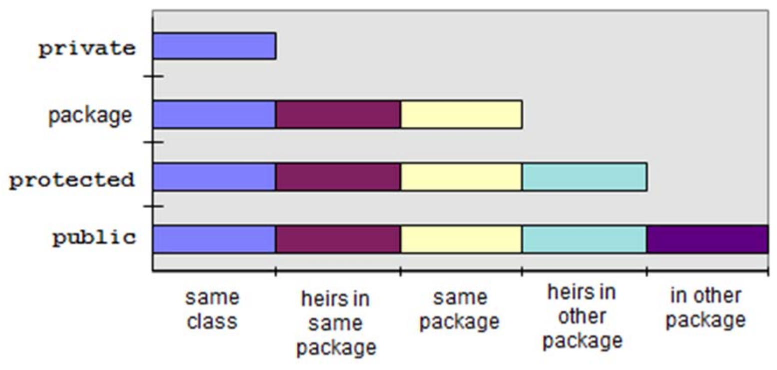
A typical OO Program

```
class point {
    protected int c;
    int getColor() { return(c); }
    int distance() { return(0); }
}
class cartesianPoint extends point{
    private int x, y;
    int distance() { return(x*x + y*y); }
}
class polarPoint extends point {
    private int r, t;
    int distance() { return(r*r); }
    int angle() { return(t); }
}
```

Encapsulation and Access Control

- In many statically typed OOPLs it is possible to declare attributes (and methods) **private** or **public** or **protected** etc.
- This has no effect on the running program, but simply means that the compiler will reject programs which violate the access-rules specified
- The control is done as part of static semantic analysis
- In dynamically types OOPLs such rules will have to be enforced at run-time

Access Privileges in Java



Dynamic Dispatch Example

```
if (x == 0) {
    p = new point();
} else if (x < 0) {
    p = new cartesianPoint();
} else if (x > 0) {
    p = new polarPoint();
}
y = p.distance();
```

Which distance method is invoked?

- Invoked Method Depends on Type of Receiver!
  - if p is a point
    - return(0)
  - if p is a cartesianPoint
    - return(x*x + y*y)
  - if p is a polarPoint
    - return(r*r)

Dynamic Lookup – dynamic dispatch Message sending

> In conventional programming,

operation (operands)

meaning of an operation is always the same

— Conventional programming add (x, y)

function add has fixed meaning

> In object-oriented programming,

object → message (arguments)

object.method(arguments)

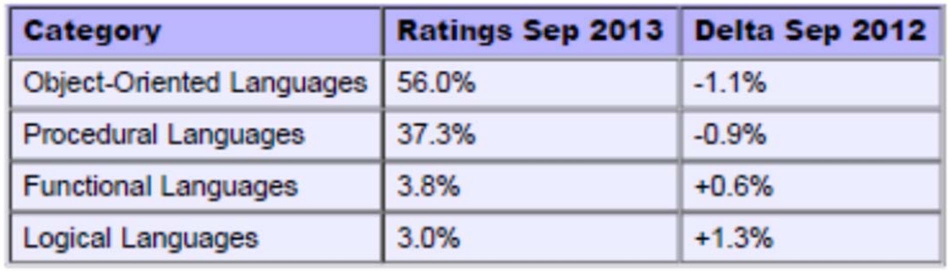
code depends on object and message

— Add two numbers x → add (y) (or x.add(y))

different add if x is integer or complex

Object Oriented Programming

> Most mainstream languages are Object Oriented:
  — Java, C#, C++, Objective C, Python, (JavaScript)

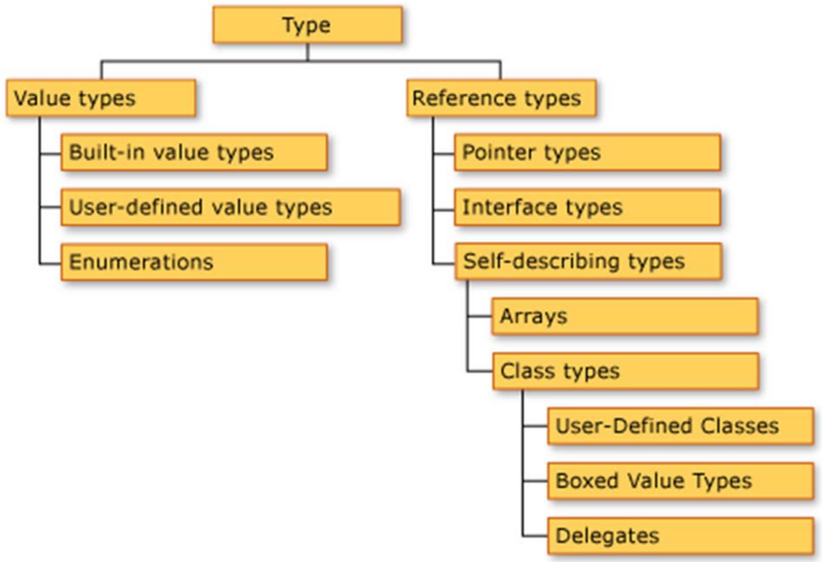


Category	Ratings Sep 2013	Delta Sep 2012
Object-Oriented Languages	56.0%	-1.1%
Procedural Languages	37.3%	-0.9%
Functional Languages	3.8%	+0.6%
Logical Languages	3.0%	+1.3%

> Lots of non OO languages now have OO features or OO dialects:
  — Fortran, Cobol, Pascal
  — (Visual)Basic, PhP
  — O'Caml, F#, Scala, Groovy, Ruby, Swift

C# - A multi paradigm language

- Object Oriented
- Single Inheritance
- Statically typed



But C# also has

- Structured Value Types
- Delegates
  - *First class anonymous functions (C# 2.0)*
  - *Lambda Expressions (C# 3.0)*
- Generics
  - *Co-and Contra-variance (C# 4.0)*
- Implicitly Typed Local Variables
- Anonymous Types
- Expression Trees
- Query Expressions
- Extension Methods
- Object Initializers
- Collection Initializers
- Iterators
- Lazy streams
- Nullable value types
- Type dymanics (C# 4.0)
- Optional and Named Parameters (C# 4.0)
- Async/await (C# 5.0)

“I invented the term *Object-Oriented* and I can tell you I did not have C++ in mind.”

Alan Kay

Who is Alan Kay?

> Alan Curtis Kay (born May 17, 1940) is an American computer scientist.

> He is best known for his pioneering work on object-oriented programming and windowing graphical user interface design, and for coining the phrase, "The best way to predict the future is to invent it."[2]

> 2003 ACM Turing award winner

> He has been elected a Fellow of the American Academy of Arts and Sciences, the National Academy of Engineering, and the Royal Society of Arts.[1]

> He is the president of the Viewpoints Research Institute, and an Adjunct Professor of Computer Science at the University of California, Los Angeles.

> He is also on the advisory board of TTI/Vanguard. Until mid-2005, he was a Senior Fellow at HP Labs, a Visiting Professor at Kyoto University, and an Adjunct Professor at the Massachusetts Institute of Technology (MIT).[3]

> After 10 years at Xerox PARC, Kay became Atari's chief scientist for three years.

> Kay is also a former professional jazz guitarist, composer, and theatrical designer, and an amateur classical pipe organist.

> Source: http://en.wikipedia.org/wiki/Alan_Kay

So what did Alan Kay have in mind?

Smalltalk

- A pure and minimal object model
  - No constructors
  - No types declaration
  - No interfaces
  - No packages/private/protected
  - No parametrized types
  - No boxing/unboxing

- Yet very powerful !

What is surprising about Smalltalk

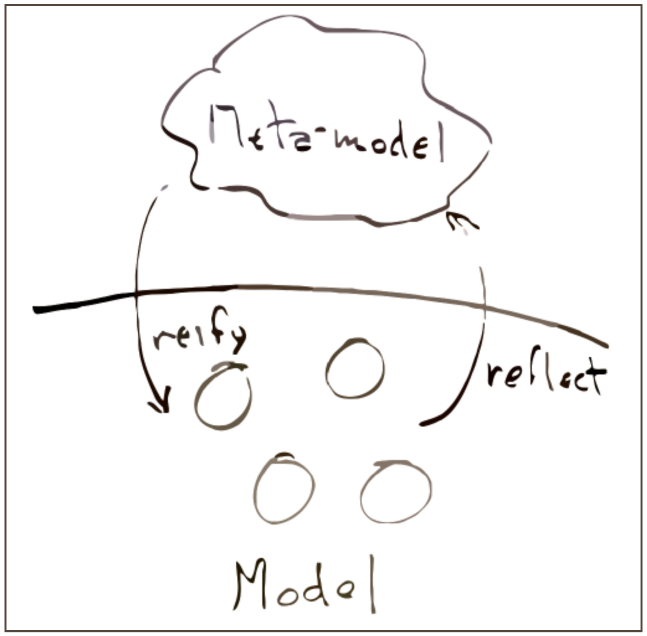
> Everything is an object
> Everything happens by sending messages

> All the source code is there all the time
> You can't lose code
> You can change everything
> You can change things without restarting the system
> The Debugger is your Friend

Everything is an object

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

Birds-eye view



Smalltalk is still today one of the few fully reflective, fully dynamic, object-oriented development environments.

We will see how a simple, uniform object model enables live, dynamic, interactive software development.

A Word of Advice

You do not have to know everything!!!

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

—Alan Knight. Smalltalk Guru

Why Smalltalk?

> *Pure* object-oriented language and environment
  — "Everything is an object"

> Origin of *many innovations* in OO development
  — RDD, IDE, MVC, XUnit …

> Improves on many of its successors
  — Fully interactive and dynamic

What is Smalltalk?

> ***Pure OO language***
> — Single inheritance
> — Dynamically typed

> ***Language and <u>environment</u>***
> — Guiding principle: *"Everything is an Object"*
> — Class browser, debugger, inspector, …
> — Mature class library and tools

> ***Virtual machine***
> — Objects exist in a persistent *image* [+ *changes*]
> — Incremental compilation

Smalltalk vs. C++ vs. Java

	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

Smalltalk: a State of Mind

> ***Small and uniform language***
>
> — Syntax fits on one sheet of paper

> ***Large library of reusable classes***
>
> — Basic Data Structures, GUI classes, Database Access, Internet, Graphics

> ***Advanced development tools***
>
> — Browsers, GUI Builders, Inspectors, Change Management Tools, Crash Recovery Tools, Project Management Tools

> ***Interactive virtual machine technology***
>
> — Truly platform-independent

> ***Team Working Environment***
>
> — Releasing, versioning, deploying

Origins of Smalltalk

> **Project at Xerox PARC in 1970s**
  — Language and environment for new generation of graphical workstations (target: “Dynabook”)

> **In Smalltalk-72, every object was an independent entity**
  — Language was designed for children (!)
  — Evolved towards a meta-reflective architecture

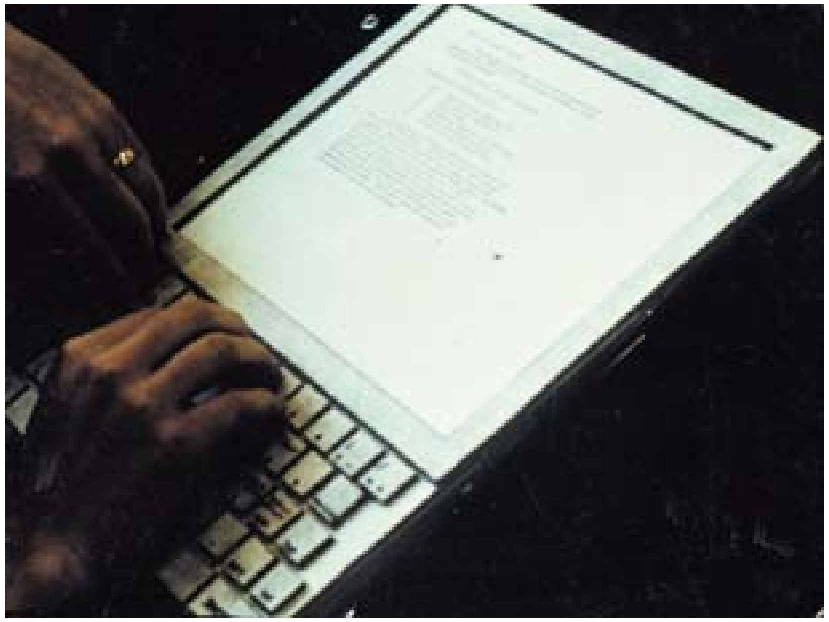
> **Smalltalk-80 is the standard**

Smalltalk — The Inspiration

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

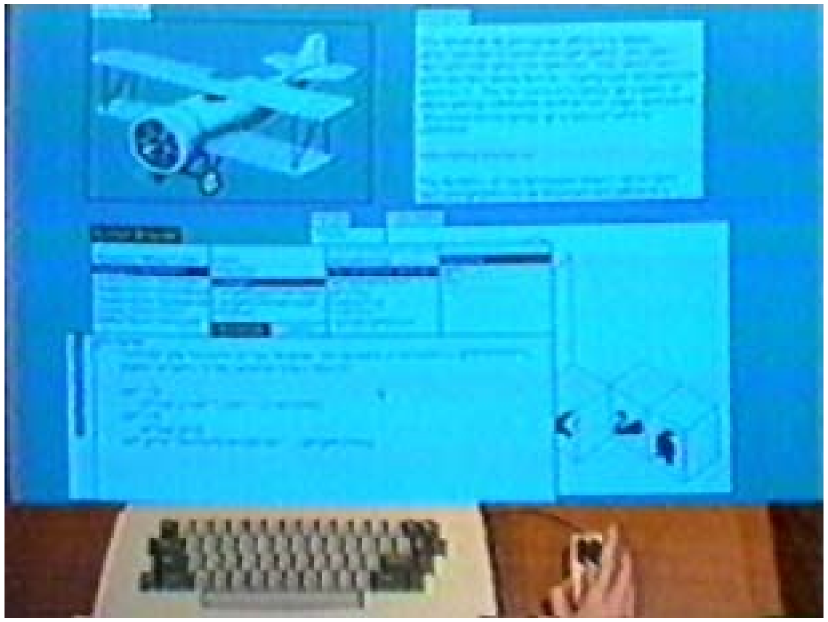
— www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Abstract.html

Dynabook Mockup



www.artmuseum.net/w2vr/archives/Kay/01_Dynabook.html

Alto: a Machine to Run Smalltalk



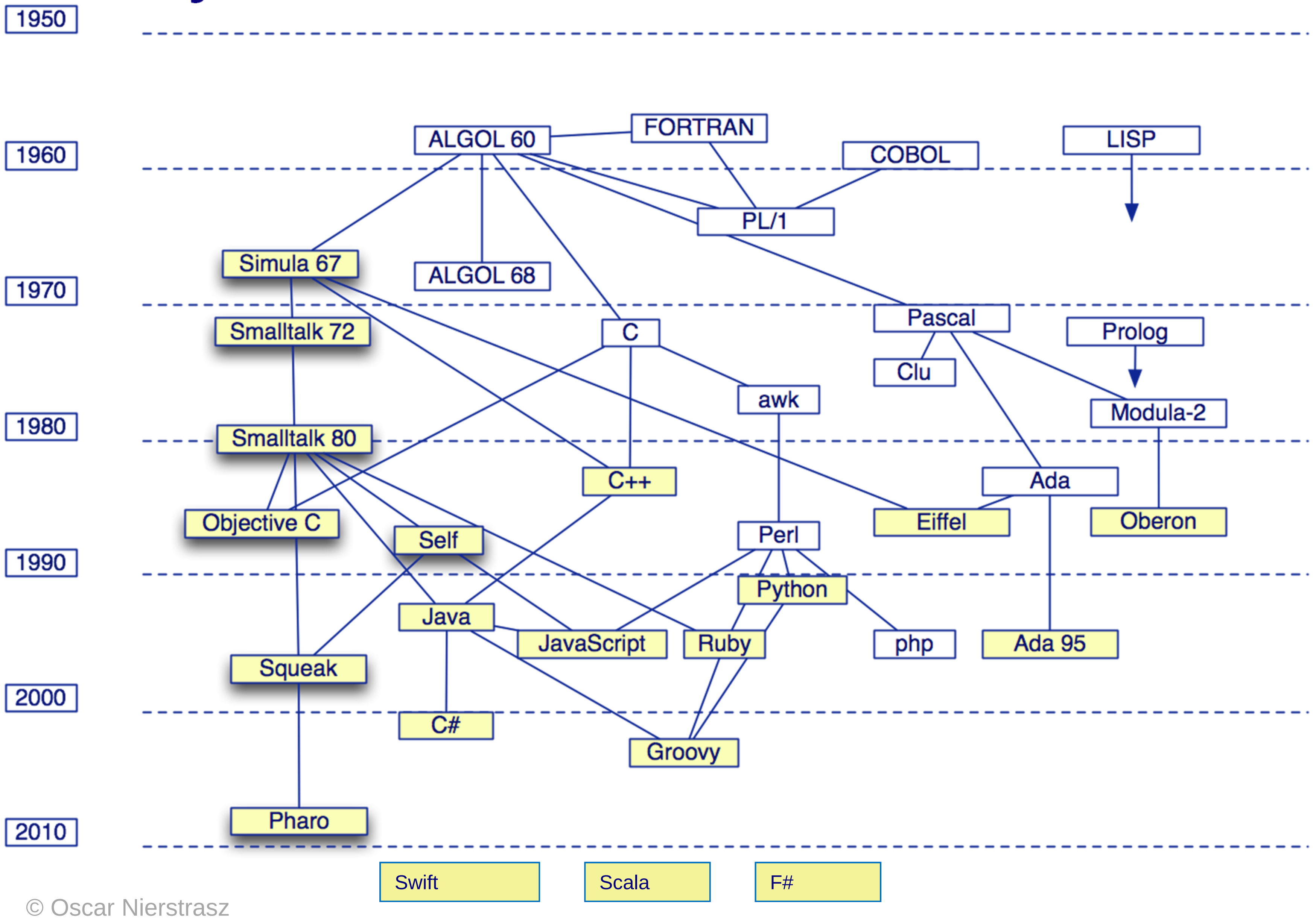
Smalltalk on Alto III

Ca. 1973/1974

Precursor, Innovator & Visionary

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

History

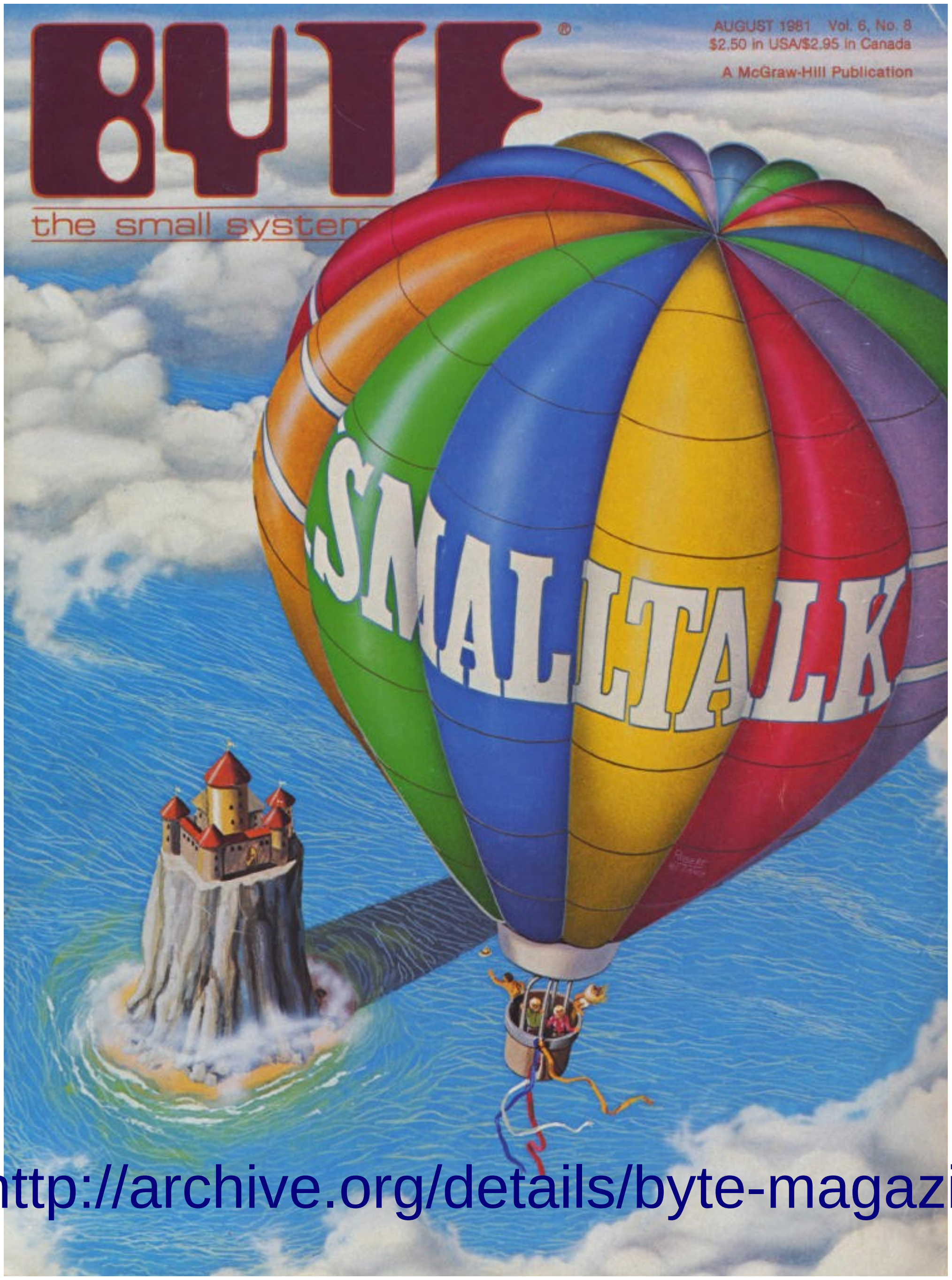

1950
1960
1970
1980
1990
2000
2010
ALGOL 60
FORTRAN
COBOL
LISP
PL/1
Simula 67
ALGOL 68
Smalltalk 72
C
Pascal
Prolog
Clu
awk
Modula-2
Smalltalk 80
C++
Ada
Objective C
Self
Perl
Eiffel
Oberon
Python
Java
JavaScript
Ruby
php
Ada 95
Squeak
C#
Groovy
Pharo
Swift
Scala
F#

The History (Internal)

> **1972 — First Interpreter**
>
> — More Agents than Objects (every object could specify its own syntax!)

> **1976 — Redesign**
>
> — A hierarchy of classes with a unique root, fixed syntax, compact bytecode, contexts, processes, semaphores, browsers, GUI library.
>
> — Projects: ThingLab, Visual Programming Environment, Programming by Rehearsal.

> **1978 — NoteTaker Project**
>
> — Experimentation with 8086 Microprocessor with only 256 KB RAM.


BYTE
the small system
AUGUST 1981 Vol. 6, No. 8
$2.50 in USA/$2.95 in Canada
A McGraw-Hill Publication
SMALLTALK

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

The History (External)

> **1980 — Smalltalk-80**
  - ASCII, cleaning primitives for portability, metaclasses, blocks as first-class objects, MVC.
  - Projects: Gallery Editor (mixing text, painting and animations) + Alternate Reality Kit (physics simulation)

> **1981 — Books + 4 external virtual machines**
  - Dec, Apple, HP and Tektronix
  - GC by generation scavenging

> **1988 — Creation of Parc Place Systems**

> **1992 — ANSI Draft**

> **1995 — New Smalltalk implementations**
  - MT, Dolphin, **Squeak**, Smalltalk/X, GNU Smalltalk

> **2000 — Fscript, GNU Smalltalk, SmallScript**

> **2002 — Smalltalk as OS: 128k ram**

What are Squeak and Pharo?

> Squeak is a modern, open-source, highly portable, fast, full-featured Smalltalk implementation
  - Based on original Smalltalk-80 code

> Pharo is a lean and clean fork of Squeak
  - www.pharo-project.org

Smalltalk — Key Concepts

- > *Everything is an object*
  - — numbers, files, editors, compilers, points, tools, booleans …
- > Everything happens by *sending messages*
- > Every object is an instance of one class
  - — which is also an object
  - — A class defines the structure and the behavior of its instances.
- > Objects have private (protected) state
  - — Encapsulation boundary is the object
- > Dynamic binding
  - — Variables are dynamically typed and bound

Objects and Classes

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

Messages and Methods

> Message — which action to perform

```
aWorkstation accept: aPacket
aMonster eat: aCookie
```

> Method — how to carry out the action

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

Example in Java

```
> public void accept(PacketType aPacket) {
>   if (aPacket.isAddressedTo(this))
        {System.out.println(”A packet is accepted by the Workstation”
  ++
          this.name.toString());}
    else
        {super.accept(aPackage);}
> }
```

Example in Java:

```
>   public void accept(PacketType aPacket) {
>     if (aPacket.isAddressedTo(this))
          {System.out.println(”A packet is accepted by the Workstation”
   ++
            this.name.toString());}
      else
          {super.accept(aPackage);}
>   }
```

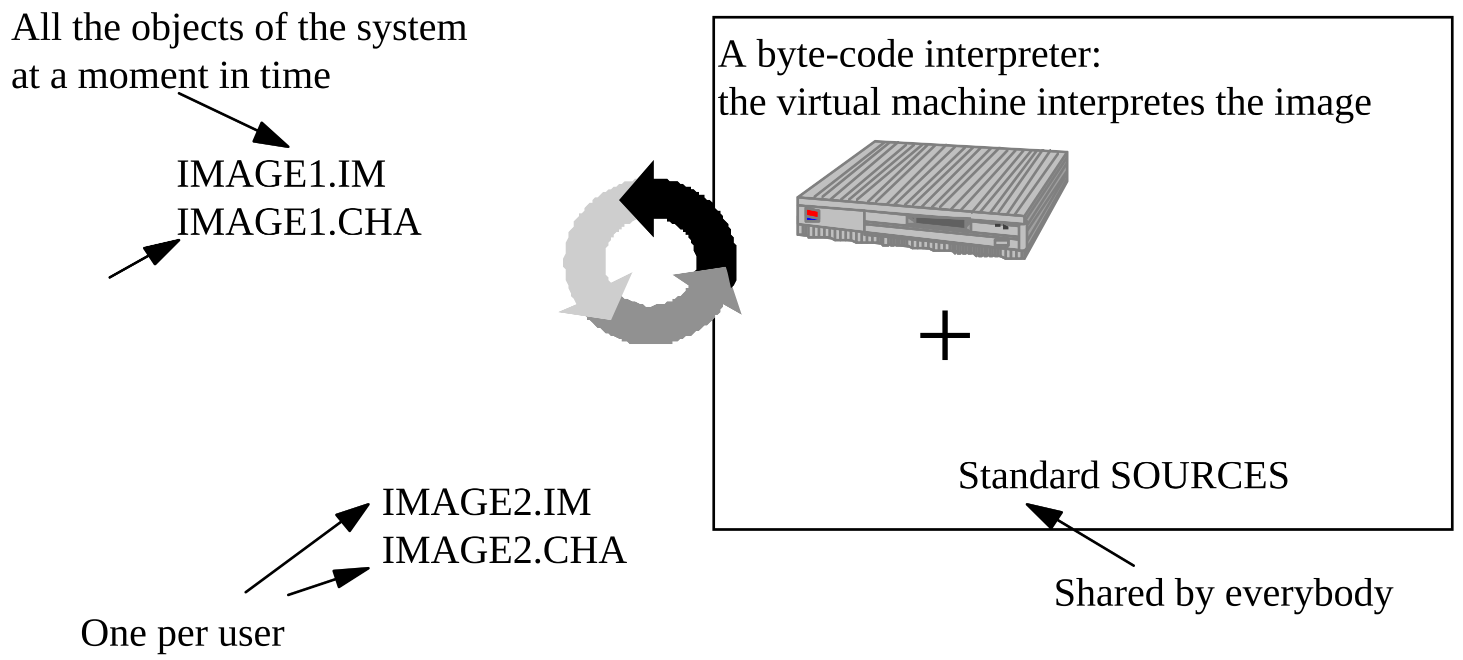
SmallTalk Syntax takes a while getting used to

- No Curly brackets in SmallTalk
- Very few syntactic elements
- Almost everything are messages to objects

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

Smalltalk Run-Time Architecture

> Virtual Machine + Image + Changes and Sources


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
IMAGE2.IM
IMAGE2.CHA
One per user
Shared by everybody

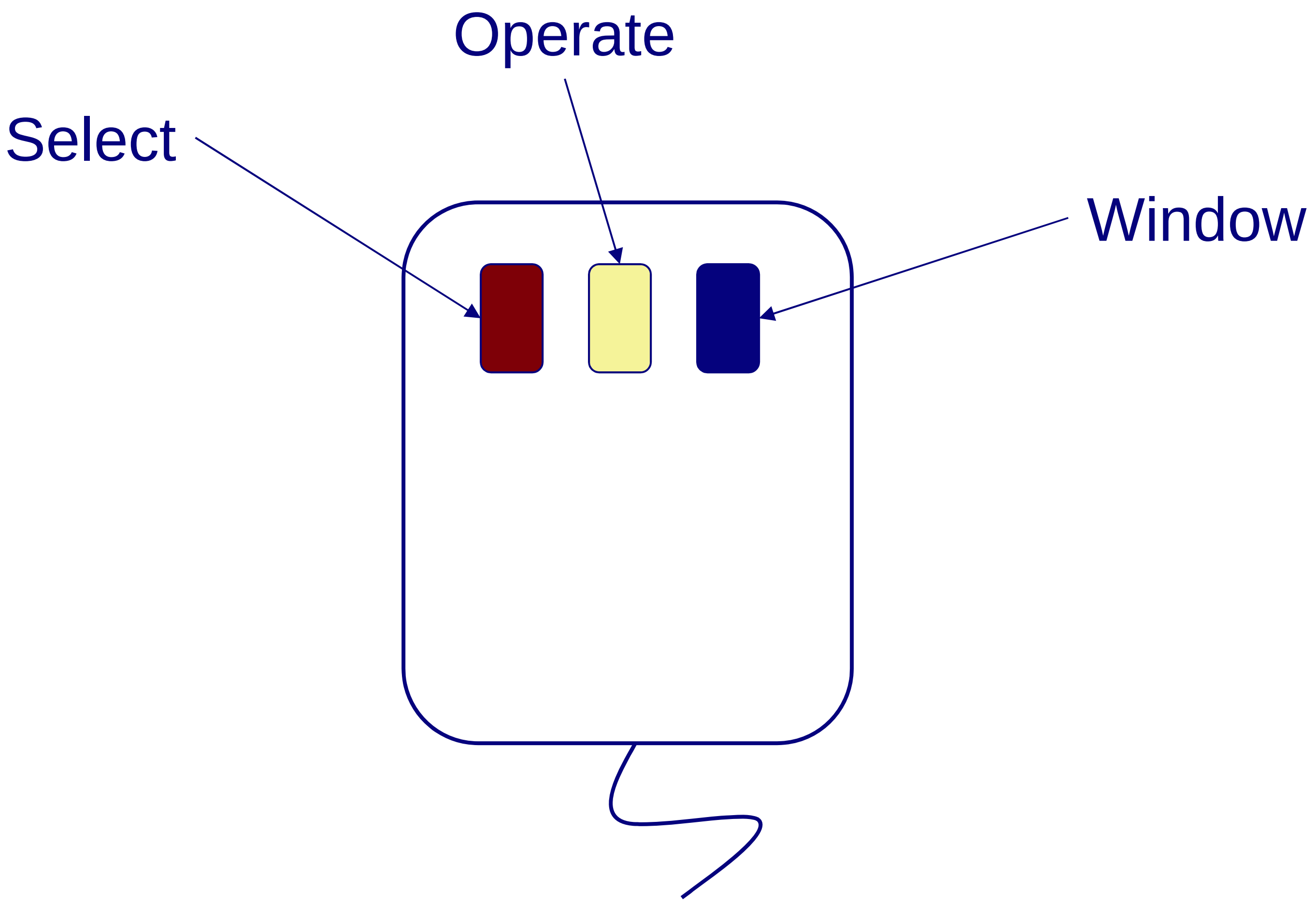
> Image = bytecodes

> Sources and changes = code (text)

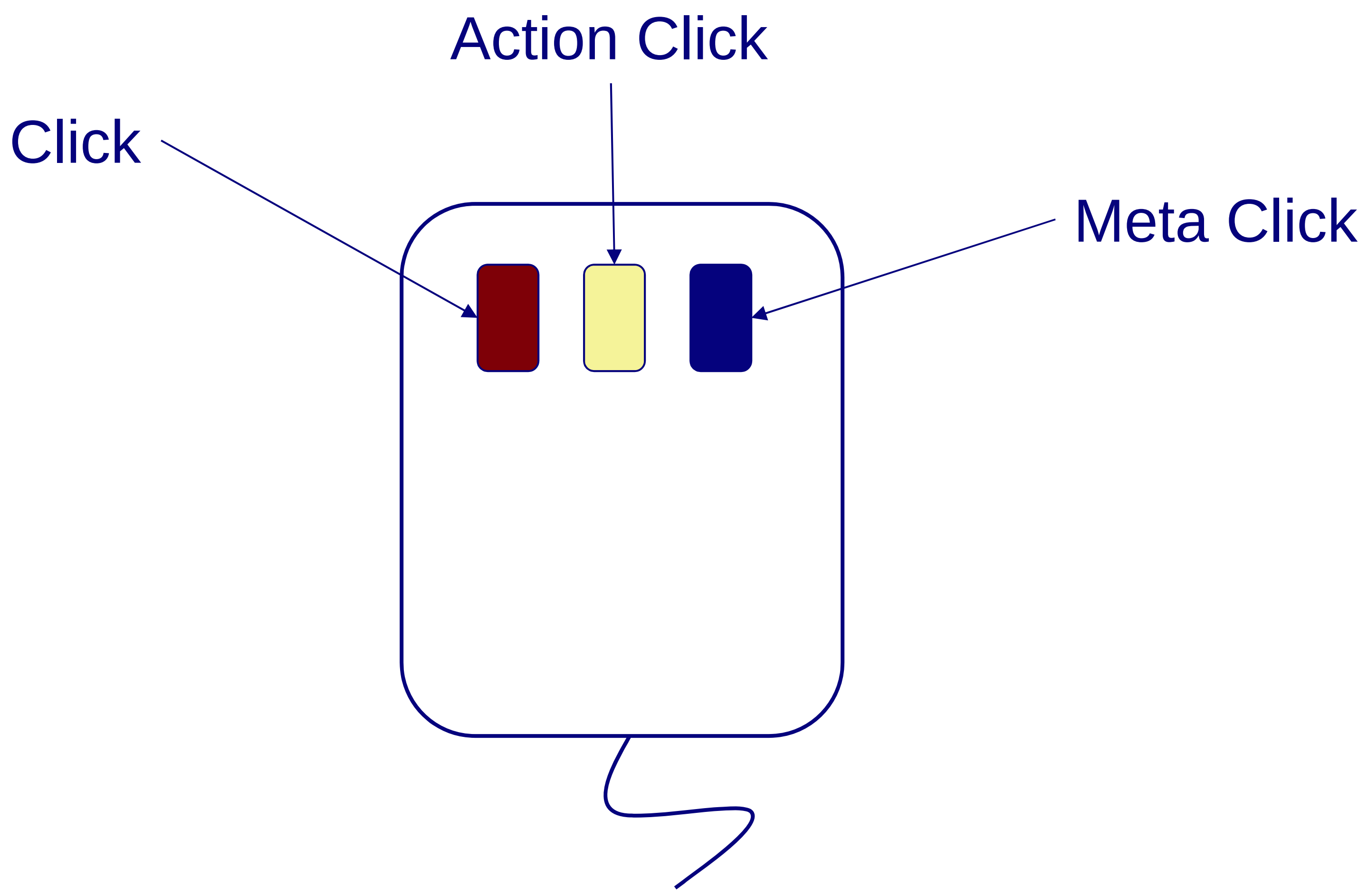
Smalltalk Run-Time Architecture

- Byte-code is translated to native code by a just-in-time compiler
  - Some Smalltalks, but not Pharo
- Source and changes are not needed to interpret the byte-code.
  - Just needed for development
  - Normally removed for deployment
- An application can be delivered as byte-code files that will be executed with a VM.
  - The development image is stripped to remove the unnecessary development components.

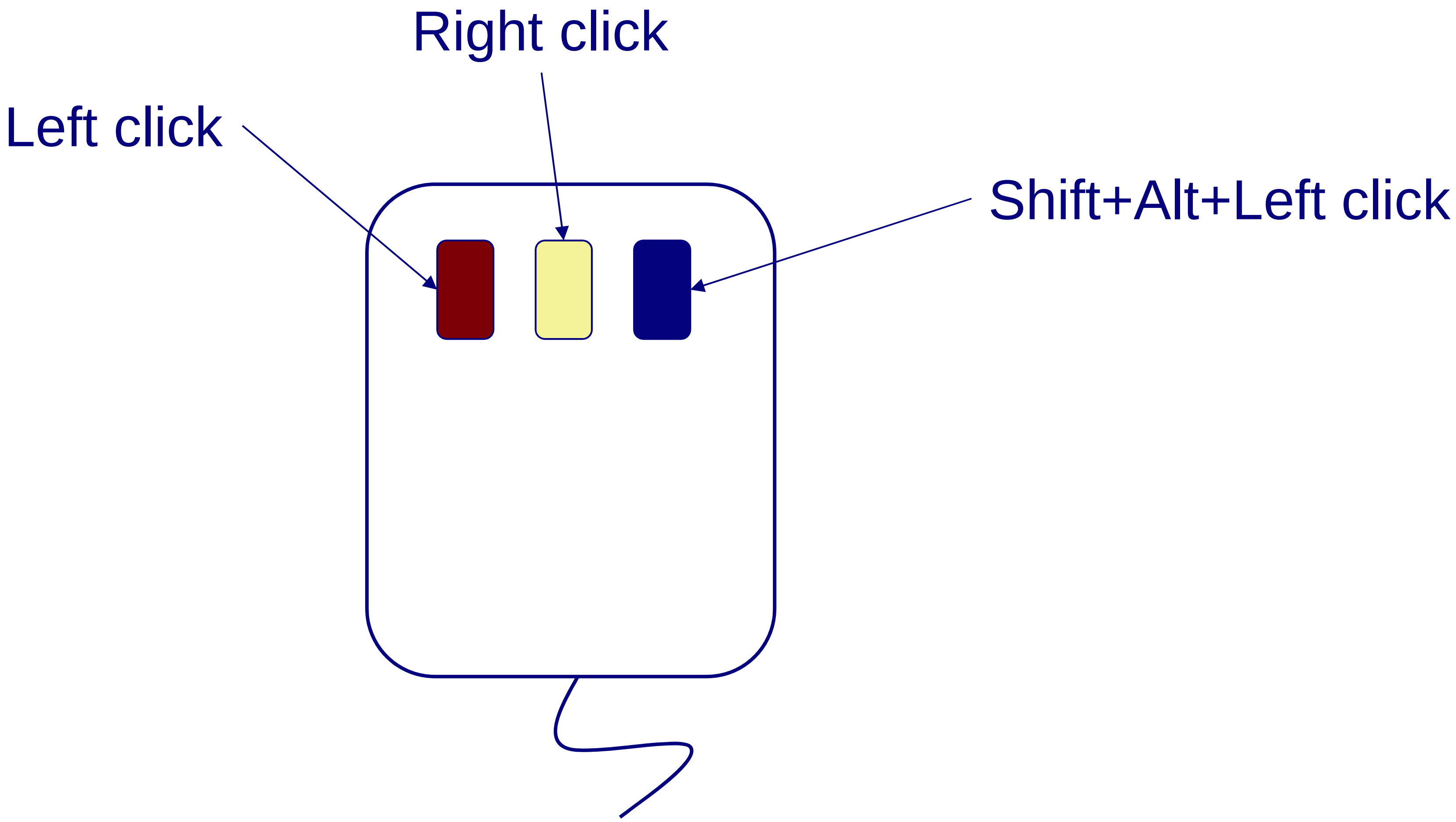
Mouse Semantics



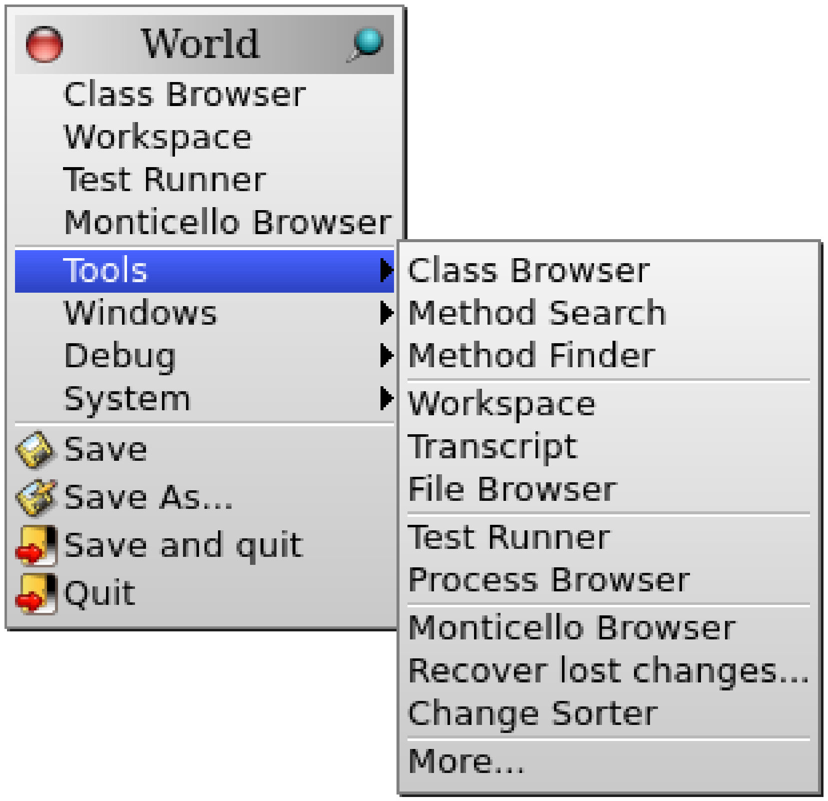
Mouse Semantics (Pharo version)



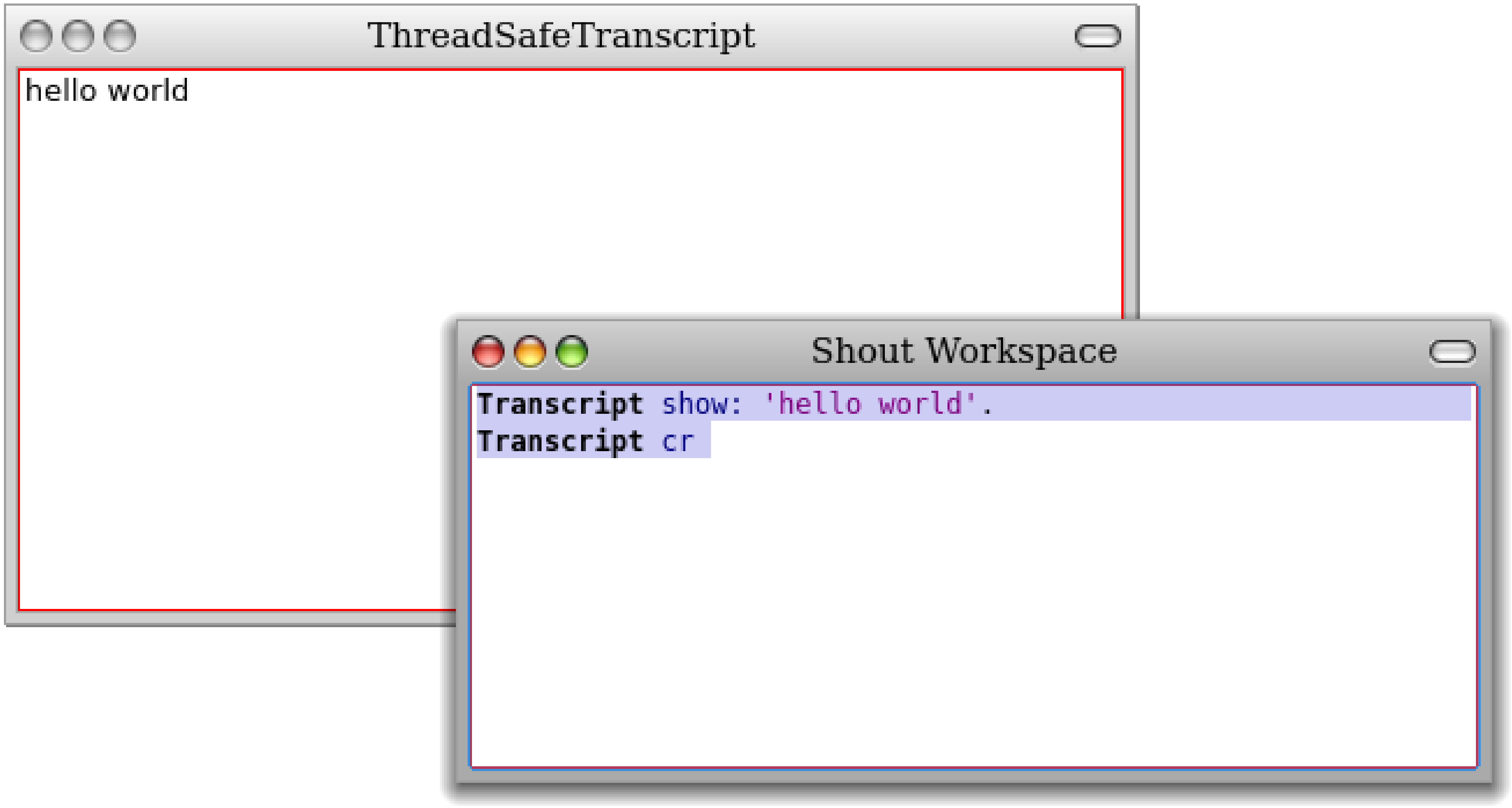
Mouse Semantics (on Windows)



World Menu


World
Class Browser
Workspace
Test Runner
Monticello Browser
Tools
Windows
Debug
System
Save
Save As...
Save and quit
Quit
Class Browser
Method Search
Method Finder
Workspace
Transcript
File Browser
Test Runner
Process Browser
Monticello Browser
Recover lost changes...
Change Sorter
More...

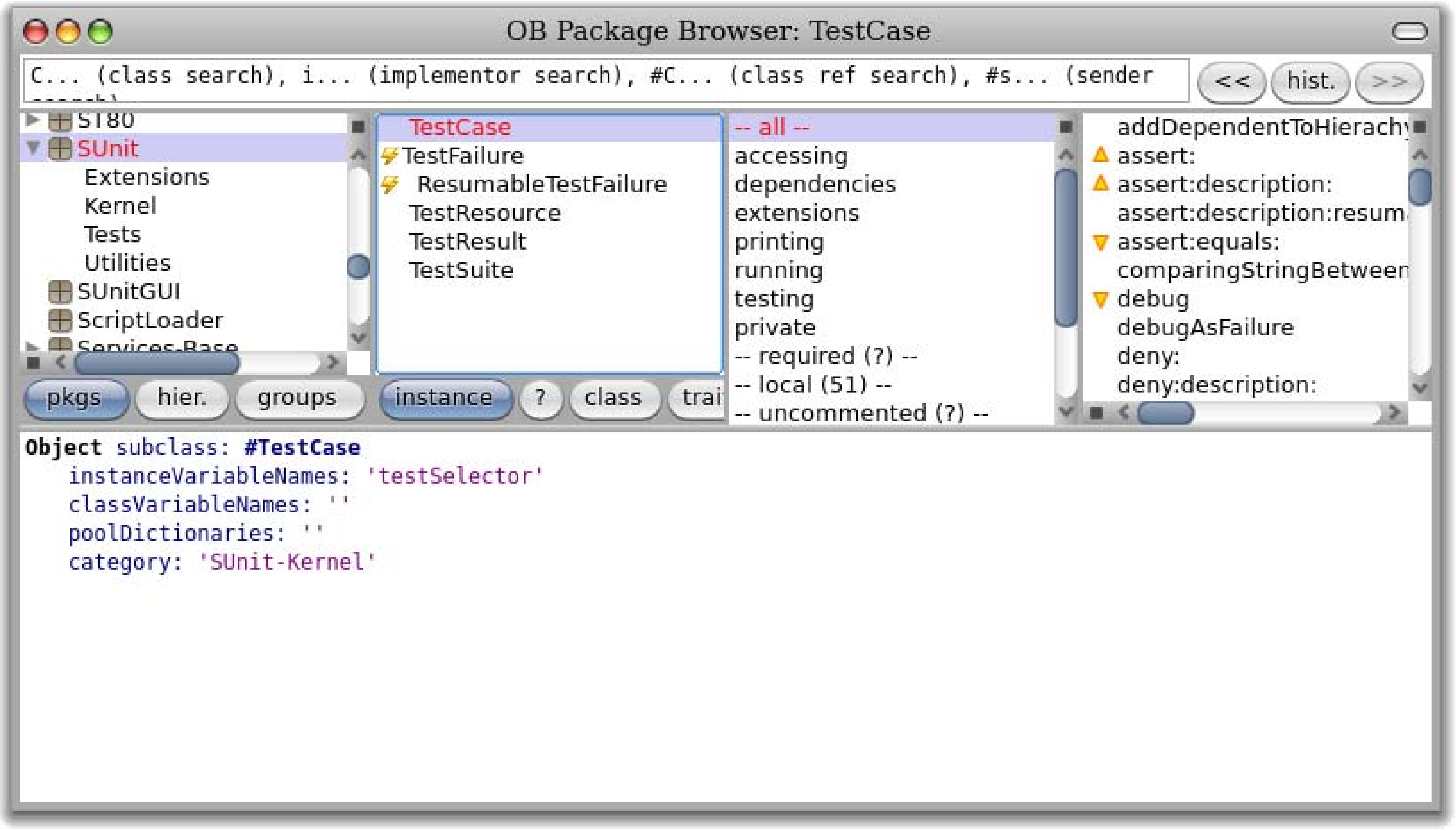
"Hello World"



```
hello world
```

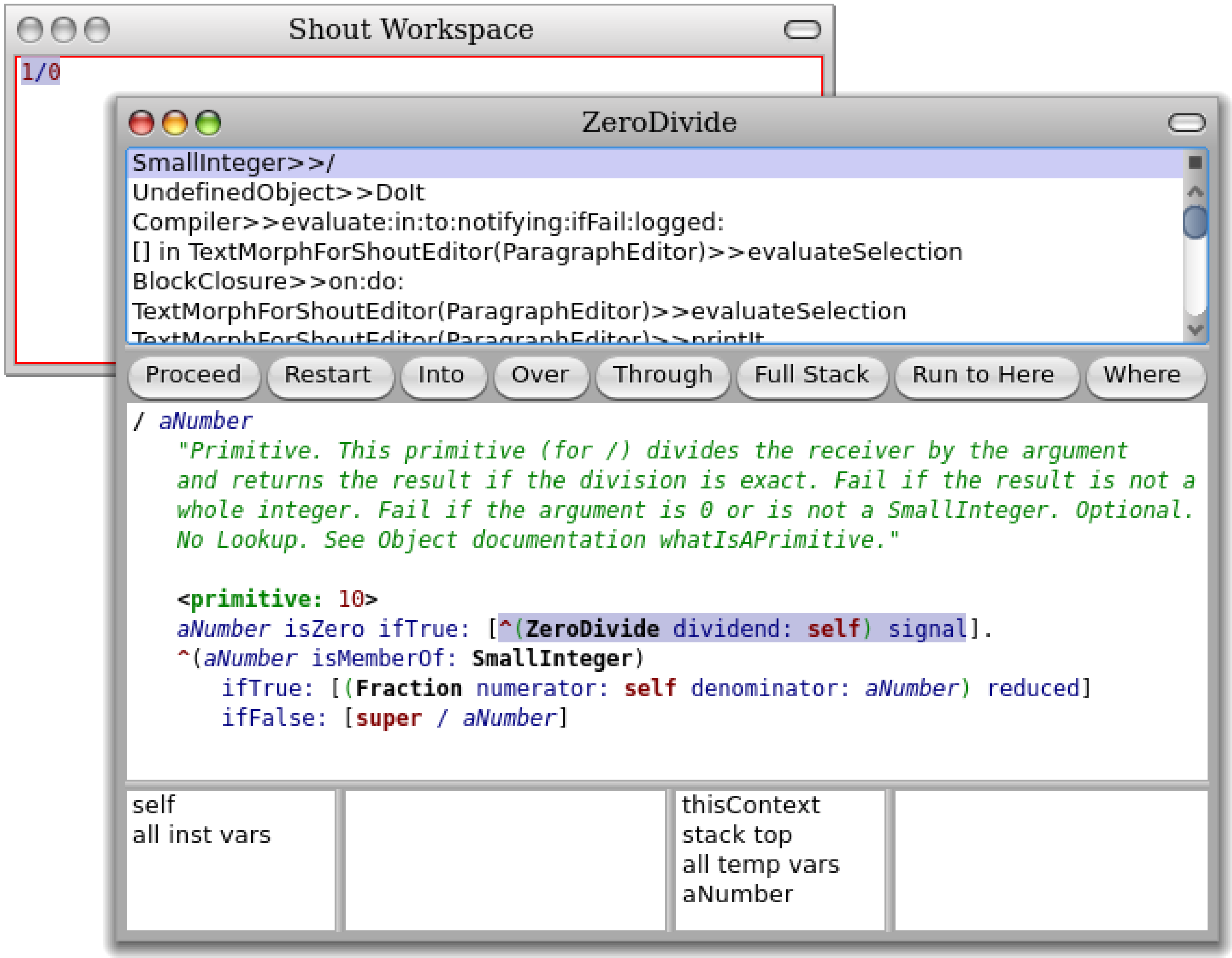
```
Transcript show: 'hello world'.
Transcript cr
```

The Smalltalk Browser



```
Object subclass: #TestCase
    instanceVariableNames: 'testSelector'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SUnit-Kernel'
```

The Debugger


Shout Workspace

```
1/0
```

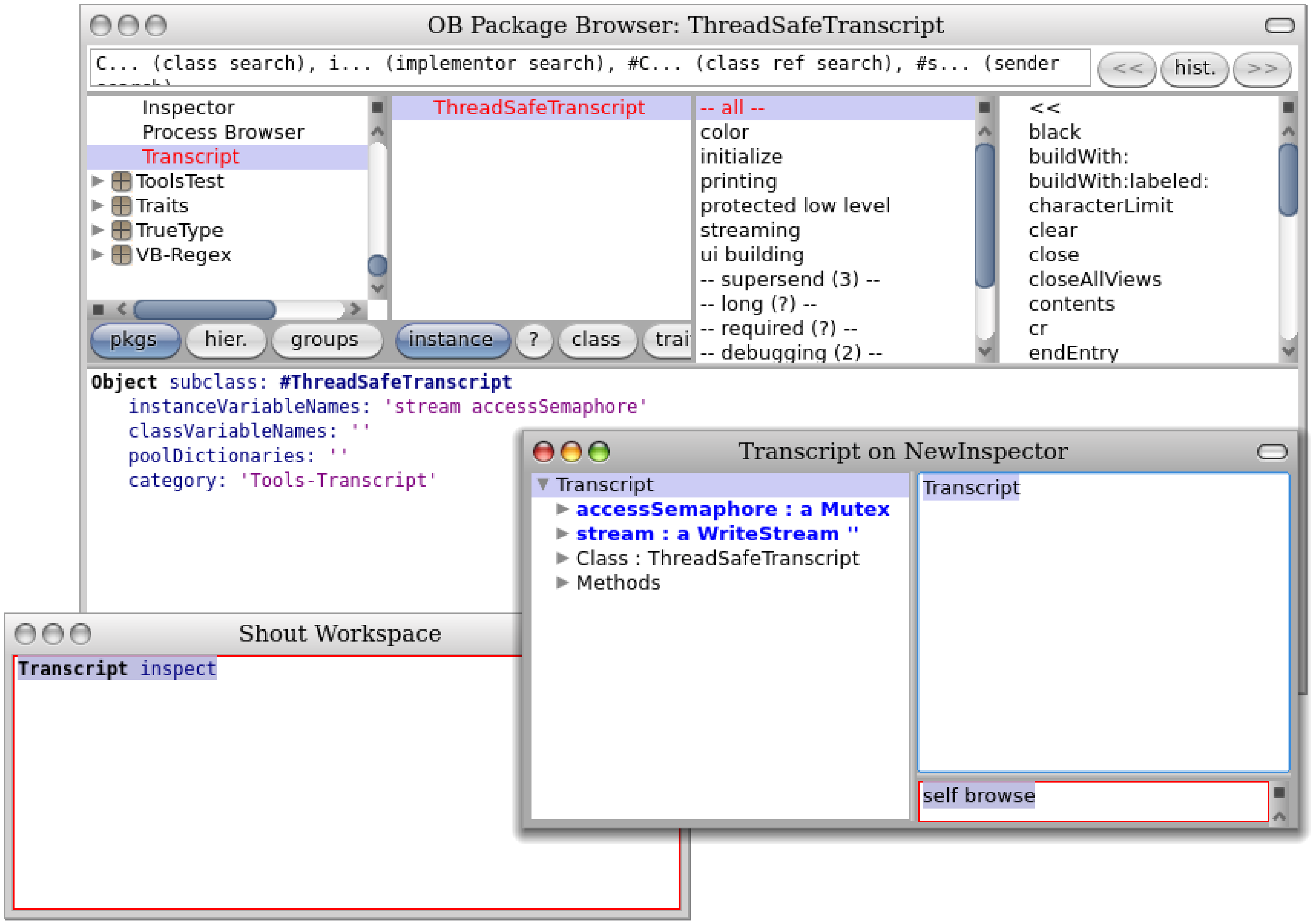
ZeroDivide

```
SmallInteger>>/
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:logged:
[] in TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
BlockClosure>>on:do:
TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
```

```
/ aNumber
    "Primitive. This primitive (for /) divides the receiver by the argument
    and returns the result if the division is exact. Fail if the result is not a
    whole integer. Fail if the argument is 0 or is not a SmallInteger. Optional.
    No Lookup. See Object documentation whatIsAPrimitive."

    <primitive: 10>
    aNumber isZero ifTrue: [^(ZeroDivide dividend: self) signal].
    ^(aNumber isMemberOf: SmallInteger)
        ifTrue: [(Fraction numerator: self denominator: aNumber) reduced]
        ifFalse: [super / aNumber]
```

The Inspector


OB Package Browser: ThreadSafeTranscript
<<
hist.
>>
pkgs
hier.
groups
instance
?
class
trai
Transcript on NewInspector
Shout Workspace

```
C... (class search), i... (implementor search), #C... (class ref search), #s... (sender search)
```

Inspector	ThreadSafeTranscript	-- all --	<<
Process Browser		color	black
Transcript		initialize	buildWith:
ToolsTest		printing	buildWith:labeled:
Traits		protected low level	characterLimit
TrueType		streaming	clear
VB-Regex		ui building	close
		-- supersend (3) --	closeAllViews
		-- long (?) --	contents
		-- required (?) --	cr
		-- debugging (2) --	endEntry

```
Object subclass: #ThreadSafeTranscript
    instanceVariableNames: 'stream accessSemaphore'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tools-Transcript'
```

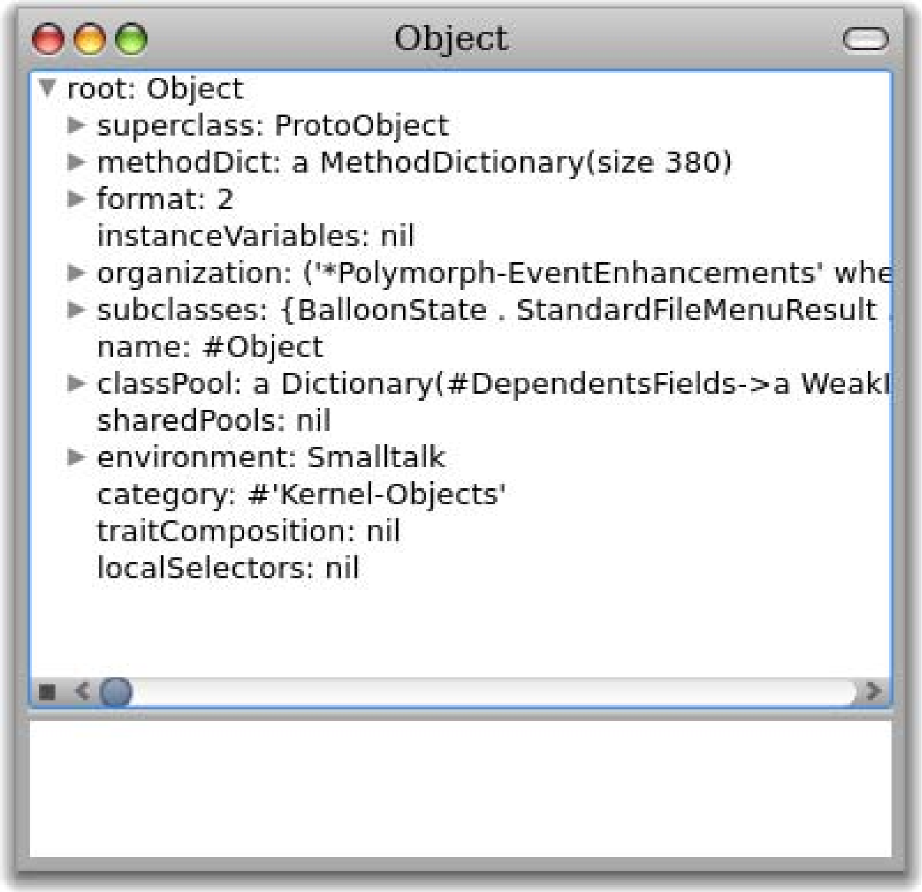
```
Transcript
    accessSemaphore : a Mutex
    stream : a WriteStream ''
    Class : ThreadSafeTranscript
    Methods
```

```
Transcript
```

```
self browse
```

```
Transcript inspect
```

The Explorer



```
▼ root: Object
   ▶ superclass: ProtoObject
   ▶ methodDict: a MethodDictionary(size 380)
   ▶ format: 2
     instanceVariables: nil
   ▶ organization: ('*Polymorph-EventEnhancements' whe
   ▶ subclasses: {BalloonState . StandardFileMenuResult
     name: #Object
   ▶ classPool: a Dictionary(#DependentsFields->a WeakI
     sharedPools: nil
   ▶ environment: Smalltalk
     category: #'Kernel-Objects'
     traitComposition: nil
     localSelectors: nil
```

Other Tools

- > File Browser
  - — *Browse, import, open files*
- > Method Finder, Message Name tool
  - — *Find methods by name, behaviour*
- > Change Sorter
  - — *Name, organize all source code changes*
- > SUnit Test Runner
  - — *Manage & run unit tests*

2. Smalltalk Basics

Birds-eye view

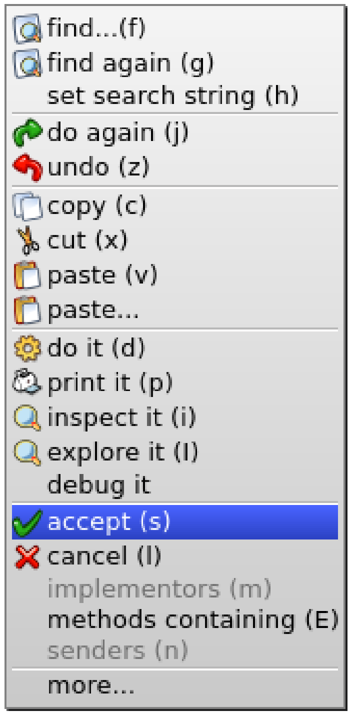
Less is More — simple syntax and semantics uniformly applied can lead to an expressive and flexible system, not an impoverished one.

Objects in Smalltalk

- *Everything* is an object
  - Things only happen by message passing
  - Variables are dynamically bound
- State is private to objects
  - “protected” for subclasses
- Methods are public
  - “private” methods by convention only
- (Nearly) every object is a reference
  - Unused objects are garbage-collected
- Single inheritance

Accept, DoIt, PrintIt and InspectIt

- > *Accept*
  - — Compile a method or a class definition
- > *DoIt*
  - — Evaluate an expression
- > *PrintIt*
  - — Evaluate an expression and print the result (`#printOn:`)
- > *InspectIt*
  - — Evaluate an expression and inspect the result (`#inspect`)

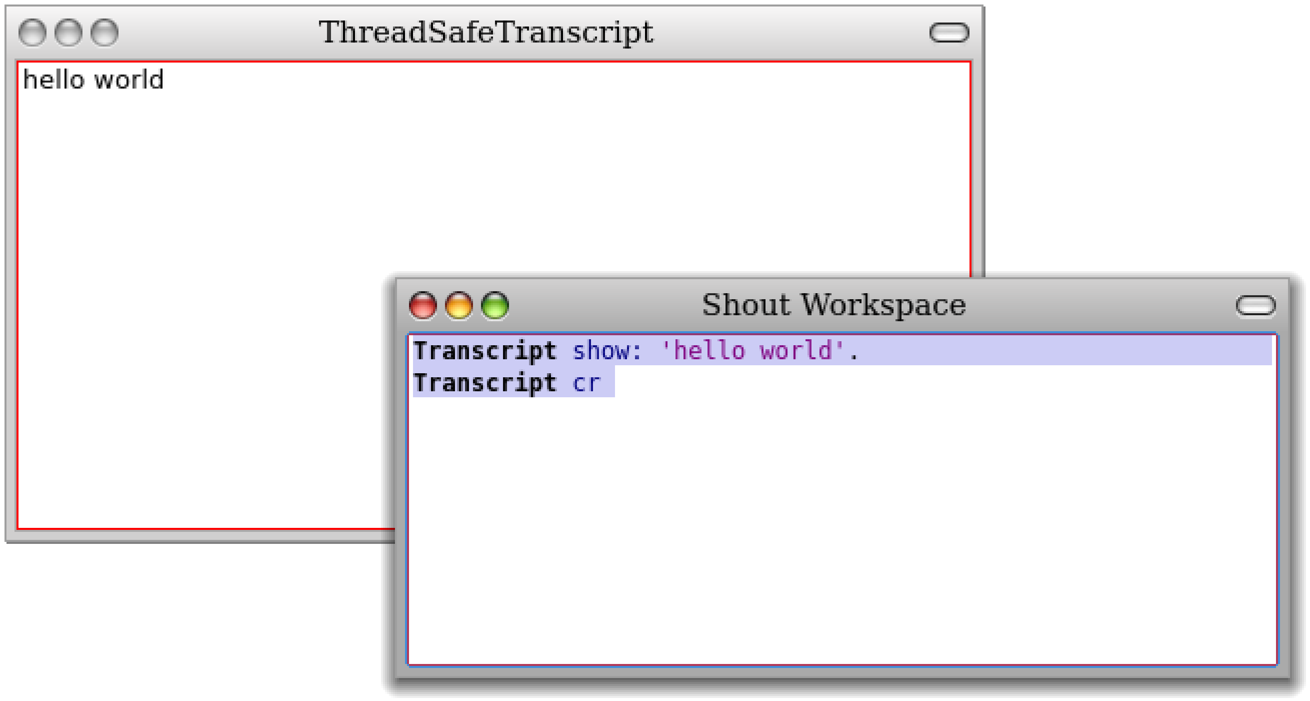


Hello World

> At anytime, *in any tool*, we can dynamically ask the system to evaluate an expression.
  — To evaluate an expression, select it and with the middle mouse button apply doIt.

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

“Hello World”



```
hello world
```

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

> Transcript is a kind of “standard output”
  — a TextCollector instance associated with the launcher.

Everything is an Object

> ***Smalltalk is a consistent, uniform world, written in itself***
  - You can learn how it is implemented, you can extend it or even modify it.
  - All the code is available and readable.
    - *The workspace is an object.*
    - *The window is an instance of SystemWindow.*
    - *The text editor is an instance of ParagraphEditor.*
    - *The scrollbars are objects too.*
    - `'hello word'` *is an instance of String.*
    - `#show:` *is a Symbol*
    - *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.*

Smalltalk Syntax on a Postcard

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

Language Constructs

`^`	return
`"…"`	comment
`#`	symbol or array
`'…'`	string
`[ ]`	block or byte array (VisualWorks)
`.`	statement separator
`;`	message cascade
`|…|`	local or block variable
`:=`	assignment (also _ or ←)
`$_`	character
`:`	end of selector name
`_e_ _r_`	number exponent or radix
`!`	file element separator (used in change sets)
`<primitive: ...>`	for VM primitive calls

Examples

comment:	`"a comment"`
character:	`$c $h $a $r $a $c $t $e $r $s $# $@`
string:	`'a nice string'`
symbol:	`#mac #+`
array:	`#(1 2 3 (1 3) $a 4)`
dynamic array (Pharo):	`{ 1 + 2 . 3 / 4 }`
Integer:	`1, 2r101`
real:	`1.5, 6.03e-34,4, 2.4e7`
fraction:	`1/33`
boolean:	`true, false`
pseudo variables	`self, super`
point:	`10@120`

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

Messages instead of keywords

- In most languages, basic operators and control constructs are defined as language constructs and keywords
- In Smalltalk, there are only *messages* sent to objects
  - `bitShift:` (>>) is just *a message sent to a number*

```
10 bitShift: 2
```

  - `ifTrue:` (if-then-else) is just *a message sent to a boolean*

```
(x>1) ifTrue: [ Transcript show: 'bigger' ]
```

  - `do:`, `to:do:` (loops) are just *messages to collections or numbers*

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

- Minimal parsing
- Language is *extensible*

Smalltalk Syntax

Every expression is a message send

> *Unary messages*

```
Transcript cr
5 factorial
```

> *Binary messages*

```
3 + 4
```

> *Keyword messages*

```
Transcript show: 'hello world'
2 raisedTo: 32
3 raisedTo: 10 modulo: 5
```

Precedence

(…) > Unary > Binary > Keyword

1. Evaluate *left-to-right*
2. Unary messages have *highest precedence*
3. Next are binary messages
4. Keyword messages have *lowest precedence*

```
2 raisedTo: 1 + 3 factorial
```

```
128
```

5. Use *parentheses* to change precedence

```
1 + 2 * 3
1 + (2 * 3)
```

```
9      (!)
7
```

Binary Messages

> Syntax:
  - aReceiver *aSelector* anArgument
  - Where *aSelector* is made up of 1 or 2 characters from:

    `+ - / \ * ~ < > = @ % | & ! ? ,`
  - Except: second character may not be `$`

> Examples:

```
2 * 3 - 5
5 >= 7
6 = 7
'hello', ' ', 'world'
(3@4) + (1@2)
2<<5
64>>5
```

More syntax

> Comments are enclosed in *double quotes*

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

> 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
```

Variables

> Declare local variables with | … |

```
| x y |
```

> Use `:=` to assign a value to a variable

```
x := 1
```

> Old fashioned assignment operator: ← (must type “_”)

Method Return

> 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`

Block closures

> Use *square brackets* to delay evaluation of expressions

```
^ 1 < 2 ifTrue: ['smaller'] ifFalse: ['bigger']
```

```
'smaller'
```

Variables

> *Local variables* within methods (or blocks) are delimited by `|var|`
> *Block parameters* 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
```

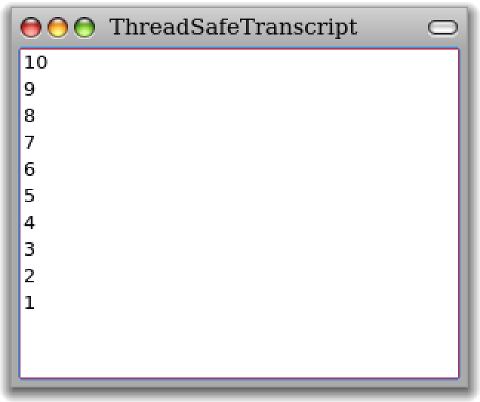
```
[:n | |x y| x := n+1. y := n-1. x * y] value: 10
```

```
99
```

Control Structures

> *Every control structure is realized by message sends*

```
|n|
n := 10.
[n>0] whileTrue:
    [ Transcript show: n; cr.
      n := n-1  ]
```



```
10
9
8
7
6
5
4
3
2
1
```

```
1 to: 10 do: [:n| Transcript show: n; cr ]
```

```
(1 to: 10) do: [:n| Transcript show: n; cr ]
```

Creating objects

> *Class methods*

```
OrderedCollection new
Array with: 1 with: 2
```

> *Factory methods*

```
1@2     "a Point"
1/2     "a Fraction"
```

Creating classes

> Send a message to a class (!)

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

Some Conventions

- > Method selector is a *symbol*, e.g., `#add:`
- > Method scope conventions using `>>`
  - — *Instance Method* defined in the class `Node`

```
Node>>accept: aPacket
```

  - — *Class Method* defined in the class `Node class` (i.e., in the class of the the class Node)

```
Node class>>withName: aSymbol
```

- > `aSomething` is an instance of the class `Something`

What you should know!

- *How can you indicate that a method is “private”?*
- *What is the difference between a comment and a string?*
- *Why does 1+2*3 = 9?*
- *What is a cascade?*
- *How is a block like a lambda expression?*
- *How do you create a new class?*
- *How do you inspect an object?*

Challenges of this Part of the Course

- > ***Mastering Smalltalk syntax***
  - — Simple, but not Java-like
- > ***Pharo Programming Environment***
  - — Requires some effort to learn at first, but worth the effort
- > ***Pharo Class Library***
  - — Need time to learn what is there
- > ***Object-oriented thinking***
  - — This is the hardest part!
- > ***Fully dynamic environment***
  - — This is the most exciting part!
- > ***Smalltalk culture***
  - — Best Practice Patterns / Test Driven Development

Can you answer these questions?

- *What ideas did Smalltalk take from Simula? From Lisp?*
- *Is there anything in Smalltalk which is not an object?*
- *If objects have private state, then how can an Inspector get at that state?*
- *How do you create a new class?*
- *What is the root of the class hierarchy?*
- *If a class is an object, then what is its class? The class of its class? …*
- *Why does Smalltalk support single (and not multiple) inheritance?*
- *Is the cascade strictly necessary?*
- *Why do you need to declare local variables if there are no static types?*

License