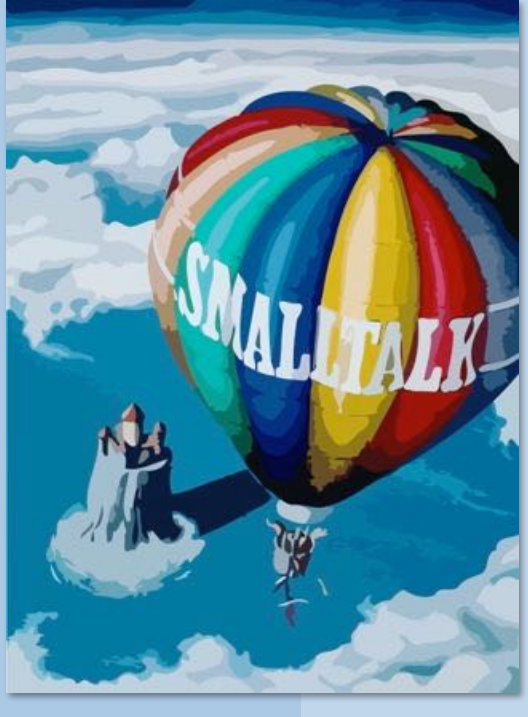
UNIVERSITÄT BERN

12. A bit of Smalltalk

Oscar Nierstrasz



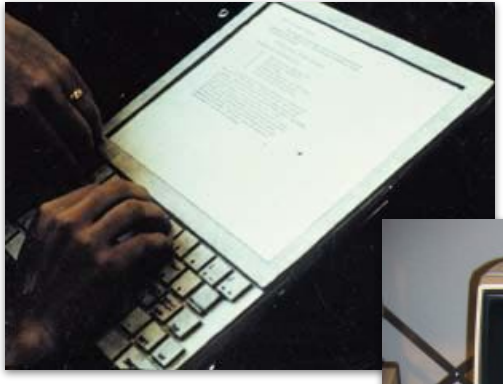
Roadmap

- The origins of Smalltalk
- What is Smalltalk?
- Syntax in a nutshell
- Seaside — web development with Smalltalk

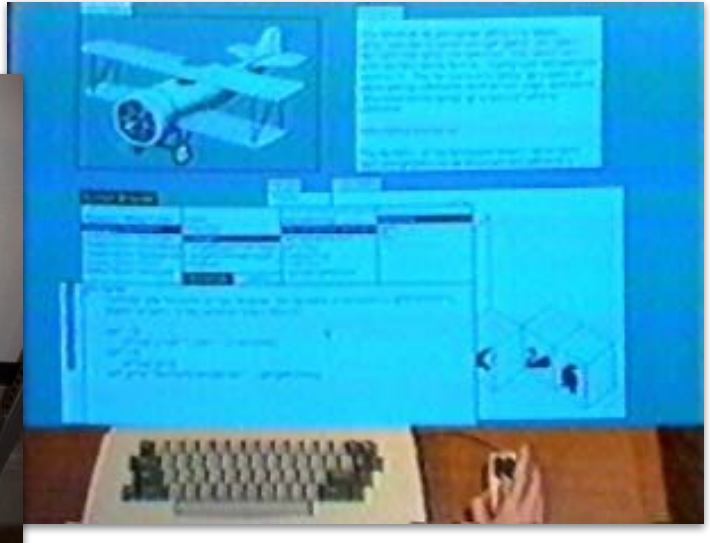
Roadmap

- **The origins of Smalltalk**
- What is Smalltalk?
- Syntax in a nutshell
- Seaside — web development with Smalltalk

The origins of Smalltalk



Alan Kay's Dynabook project (1968)



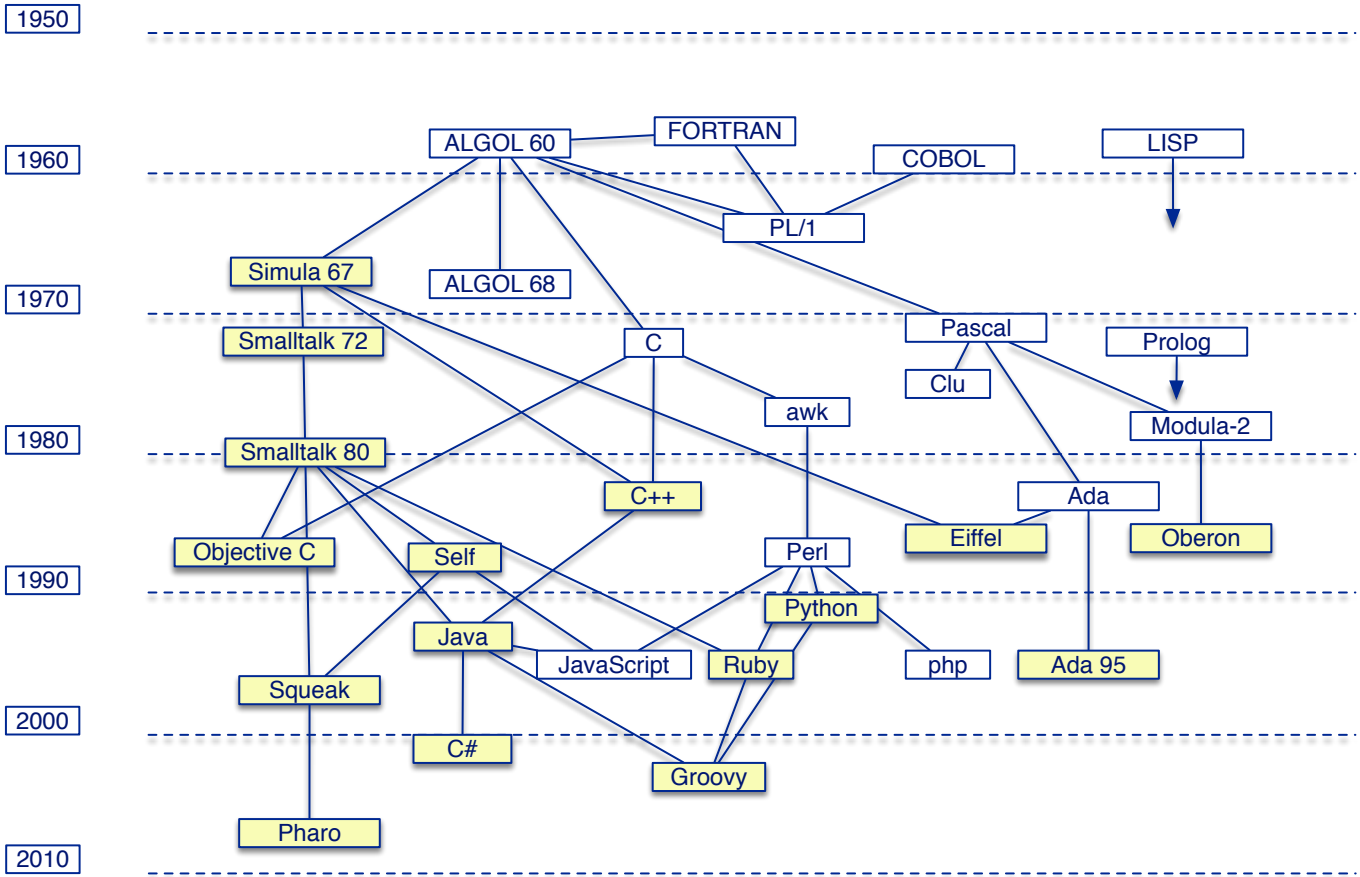
Alto — Xerox PARC (1973)

gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

In the late 60s, Alan Kay predicted that in the foreseeable future handheld multimedia computers would become affordable. He called this a “Dynabook”. (The photo shows a mockup, not a real computer.)

He reasoned that such systems would need to be based on object from the ground up, so he set up a lab at the Xerox Palo Alto Research Center (PARC) to develop such a fully object-oriented system, including both software and hardware. They developed the first graphical workstations with windowing system and mouse.

Object-oriented language genealogy



Simula was the first object-oriented language, designed by Kristen Nygaard and Ole Johan Dahl. Simula was designed in the early 60s, to support simulation programming, by adding classes and inheritance to Algol 60. The language was later standardized as Simula 67. Programmers quickly discovered that these mechanisms were useful for general-purpose programming, not just simulations.

Smalltalk adopted the ideas of objects and message-passing as the core mechanisms, not just add-ons to a procedural language.

Stroustrup ported the ideas of Simula to C to support simulation programming. The resulting language was first called “C with classes”, and later C++.

Cox added Smalltalk-style message-passing syntax to C and called it “Objective-C”.

Java integrated implementation technology from Smalltalk and syntax from C++.

Squeak and Pharo are modern descendants of Smalltalk-80.

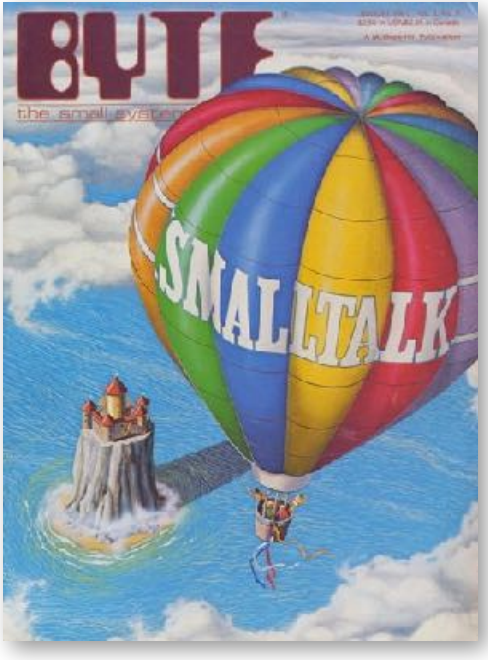
Smalltalk vs. Java vs. C++

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

The most important difference between Smalltalk, Java and C++, is that Smalltalk supports “live programming”. Whereas in Java and C++ you must first write source code and compile it before you run anything, in Smalltalk you are always programming in a live environment. You incrementally add classes and compile methods within a running system.

As a consequence, Smalltalk has to be fully reflective, allowing you to reify (“turn in objects”) all aspects of the system, and change them at run time. The only thing you cannot change from within Smalltalk is the virtual machine.

Smalltalk-80 and Pharo



Everything is an object.
Everything is there, all the time.
First windowing system with mouse.
First graphical IDE.

Smalltalk-80 was introduced to the world in 1981 in a now-famous issue of Byte Magazine. The “Smalltalk balloon” refers to this issue.

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

What are Squeak and Pharo?

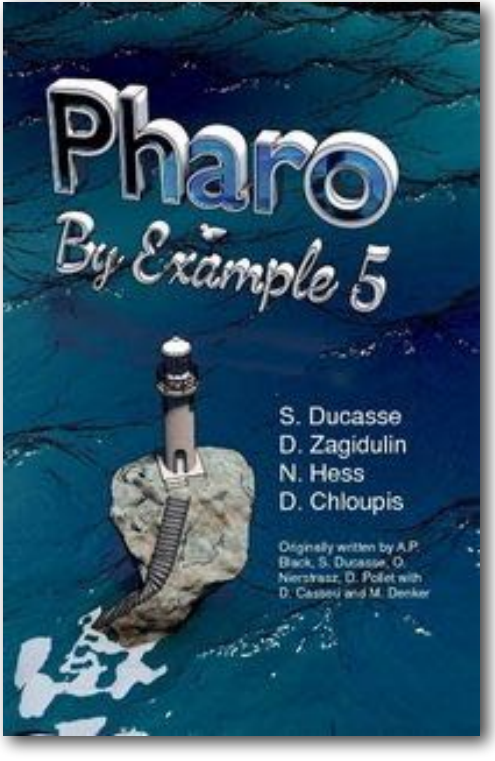
- Squeak is a modern, open-source, highly portable, fast, full-featured Smalltalk implementation
  - —Based on original Smalltalk-80 code

- Pharo is a lean and clean fork of Squeak
  - —pharo.org

Squeak was developed by members of the original Smalltalk team, with the goal of supporting and experimenting with advanced multimedia systems. The Squeak image was ported from a Smalltalk-80 image.

Pharo is a modern descendent of Smalltalk-80, largely obeying the original syntax and design, but with numerous improvements to the language, the tools, and the environment. Whereas Squeak was developed with the goal to support experimentation, Pharo aspired to offer a clean and stable platform upon which both industrial and research projects can build.

Pharo by Example



http://files.pharo.org/books/

- Free download
- Open-Source
- Print-on-demand

Pharo by Example is a free book, originally published in 2009, to teach new users Pharo Smalltalk in an example-driven way.

Pharo by Example 5 is a major revision that reflects many changes in Pharo since then.

Please read the beginning of this book for a quick introduction to Smalltalk.

Numerous related books are available online:

http://files.pharo.org/books/

All the material is hosted on github:

https://github.com/SquareBracketAssociates/

Don’t panic!

New Smalltalkers often think they need to understand all the details of a thing before they can use it.

Try to answer the question

“How does this work?”

with

“I don’t care”.

Alan Knight. Smalltalk Guru

This slide is a paraphrase of:

> 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”.
>
> alanknightsblog.blogspot.ch

Roadmap

- The origins of Smalltalk
- **What is Smalltalk?**
- Syntax in a nutshell
- Seaside — web development with Smalltalk

Two rules to remember

Everything is an object

(Nearly) everything in Smalltalk is an object, which means that you can “grab it” and talk to it. Everything that you see on the screen is an object, so you can interact with it programmatically.

The implementation of Smalltalk itself is build up of objects, so you can grab these objects and explore them. In particular, all the tools are objects, but also classes and methods are objects. This feature is extremely powerful and leads to a style of programming that is different from the usual edit/compile/run development cycle.

Everything happens by sending messages

The only way to make anything happen is by sending messages. To ask “what can I do with this object?” is the same as asking “what messages does it understand?”

The terminology of “message sending” is perhaps unfortunate, as those new to Smalltalk often assume it has something to do with network communication, but one should understand it as a metaphor: you do not “call an operation” of an object, but you politely ask it to do something by sending it a request (a “*message*”). The object then decides how to respond by checking to see if its class has a “*method*” for handling this request. If it does, it performs the method. If not, it asks its superclass if it has such a method, and so on. If this search fails, the object does not understand the message (but let’s not get into that now!).

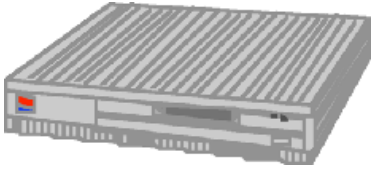
Smalltalk — a *live* programming environment

Image

+

Changes

Virtual machine



+

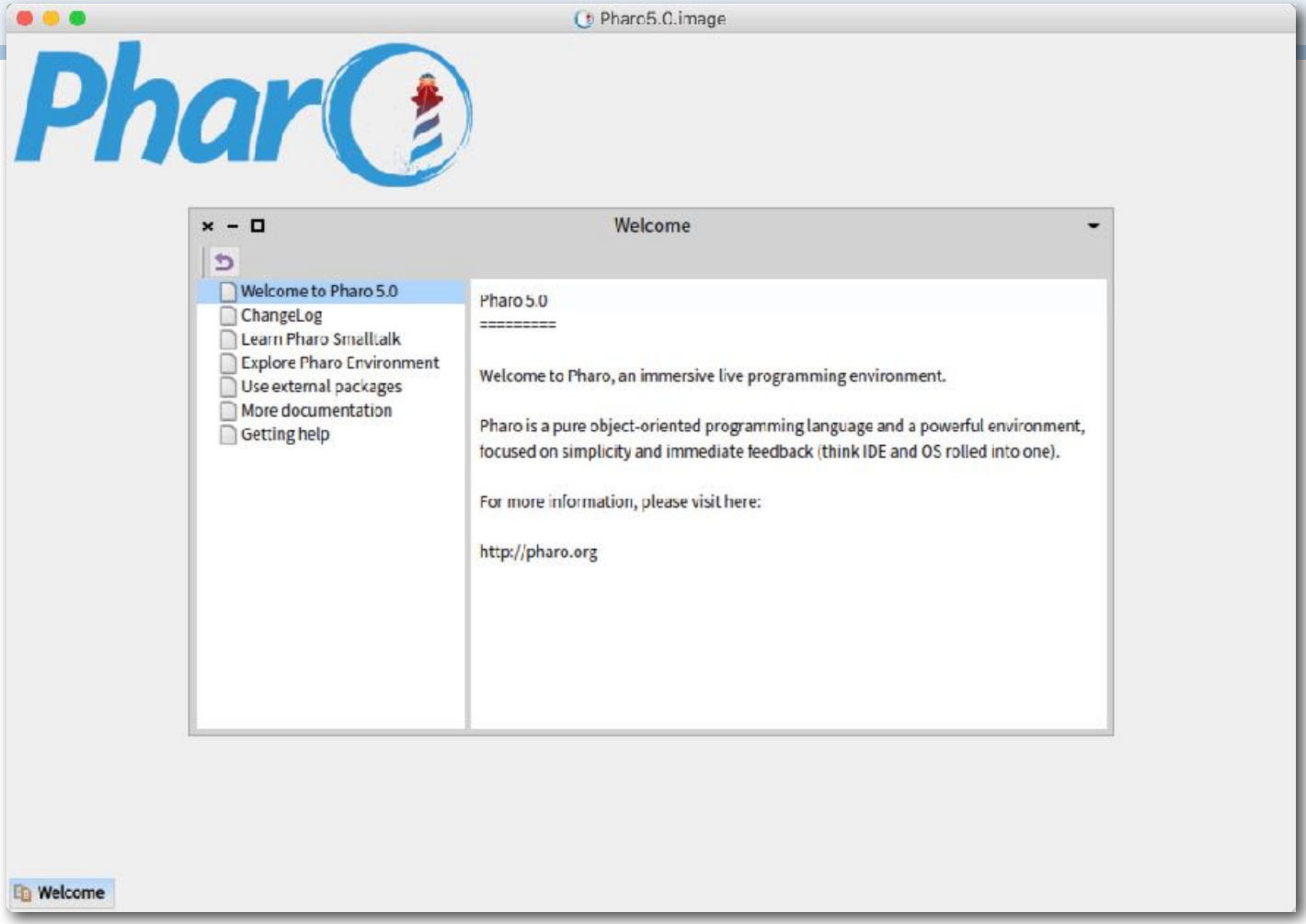
Sources

Smalltalk is often bundled into a single, “one-click” application, but there are actually four pieces that are important to understand.

Every user of Smalltalk can work with one or more Smalltalk images. The *image* file contains a snapshot of all the objects of the running system. Every time you quit Smalltalk, you can save and update this snapshot. In addition, the *changes* file consists of a log of all changes to the source code of that image, i.e., all new or changed classes and all compiled methods. If your image crashes (which is possible since Smalltalk allows you to do anything, even if that might be fatal), you can restart your image and *replay your changes*, so nothing is lost.

In addition, the virtual machine and sources files may be shared between users. The *VM* runs the bytecode of compiled methods and manages the image and changes file. Finally the *sources* file (optional) contains all the source code of objects in the base image (so you can not only explore this but modify it if you want).

Demo: Running Pharo

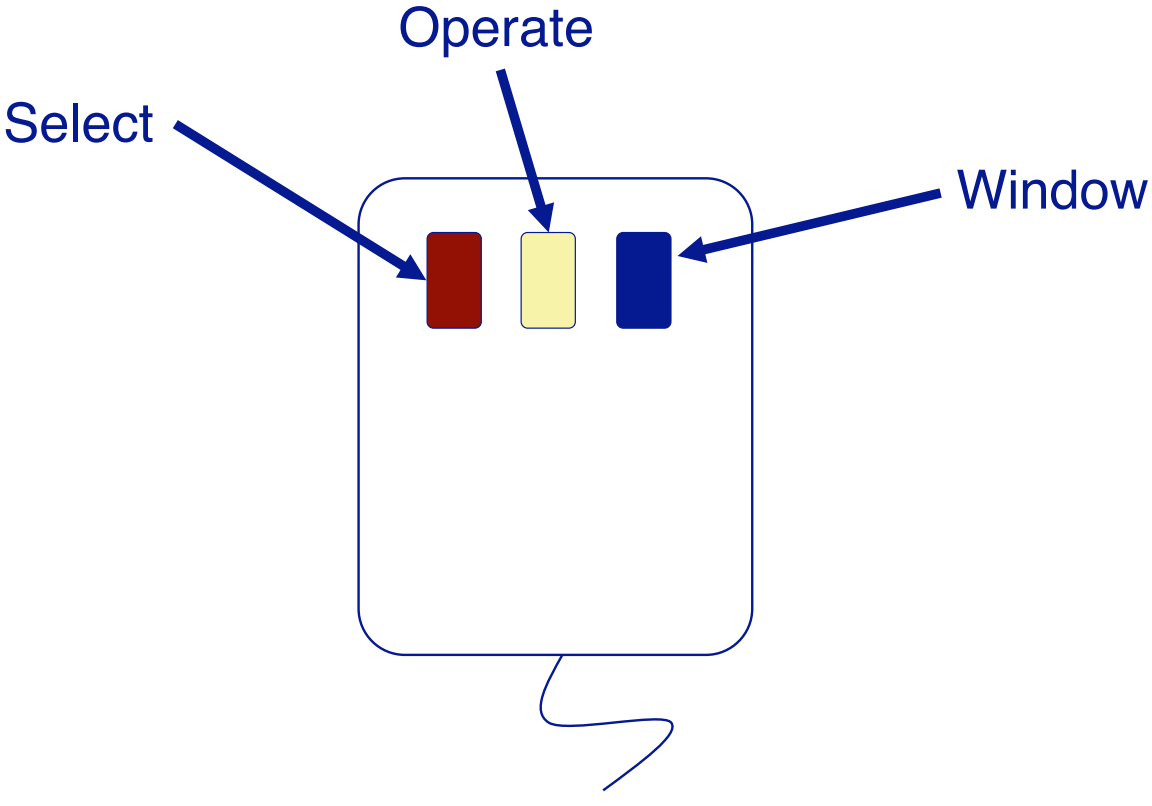


You can find the demo script in the p2-Smalltalk folder of the P2 examples repo:

```
git clone git://scg.unibe.ch/lectures-p2-examples
```

The demo illustrates: the workspace and basic tools, navigating between objects and their source code, inspection of live objects, test-driven development, debugging a live system …

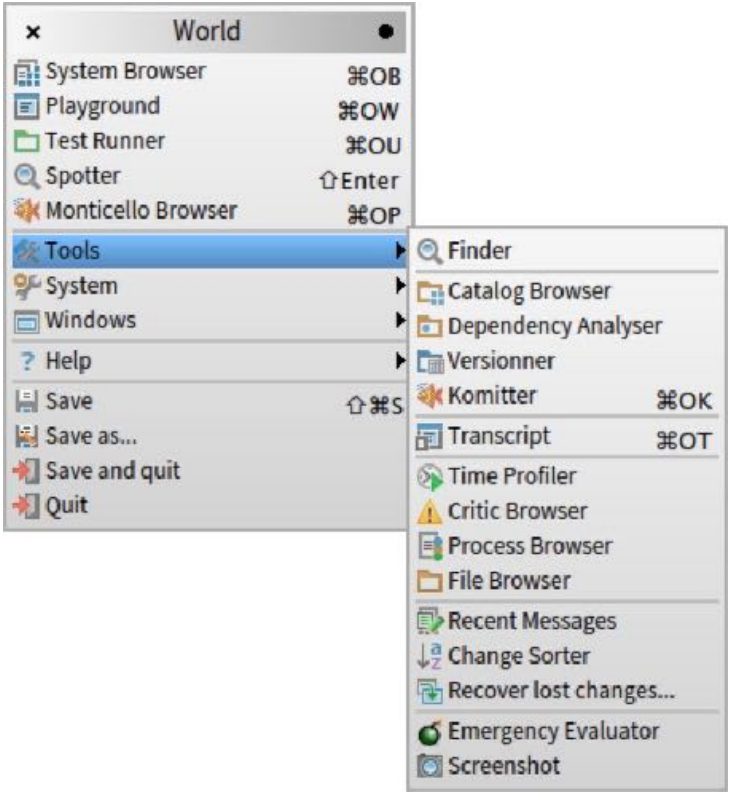
Mouse Semantics



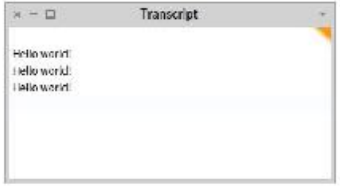
Smalltalk-80 assumes you have a “three-button mouse”. The “red button” allows you to select an object on the screen, the “yellow button” pops up a menu of things you can do with it, and the “blue button” offers a “meta menu” of system operations. (In Pharo, the “World menu”.)

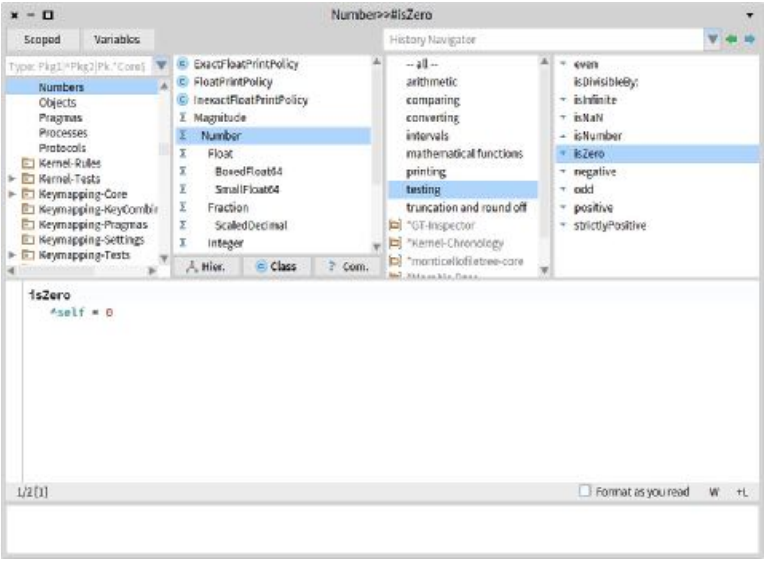
Depending on the version of Smalltalk you are using and the platform you are on, the three buttons may be mapped to different modifier keys (e.g., <ALT>, <CTL> etc.)

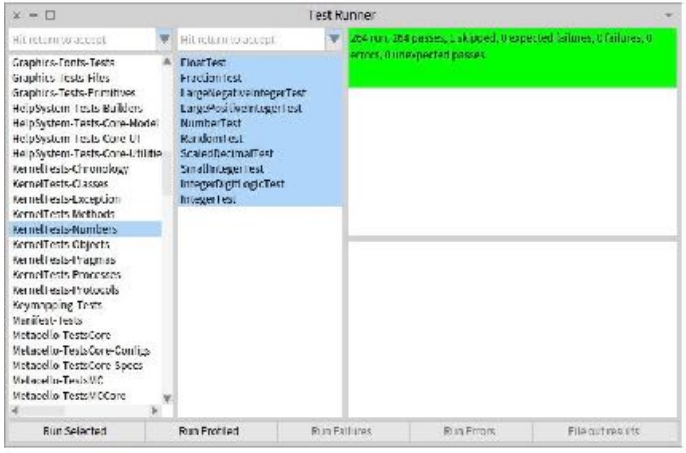
World Menu

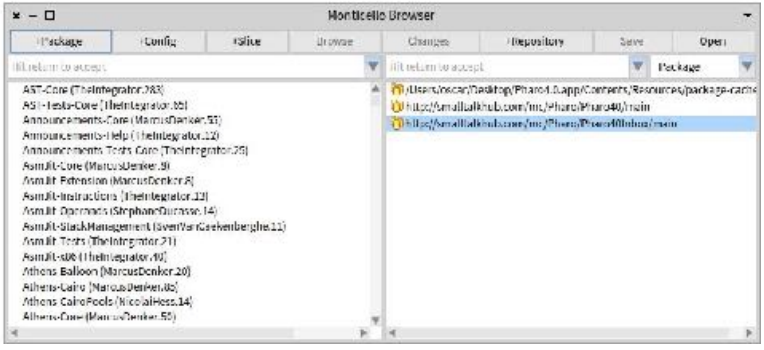

World
System Browser ⌘OB
Playground ⌘OW
Test Runner ⌘OU
Spotter ⇧Enter
Monticello Browser ⌘OP
Tools
System
Windows
Help
Save ⇧⌘S
Save as...
Save and quit
Quit
Finder
Catalog Browser
Dependency Analyser
Versionner
Komitter ⌘OK
Transcript ⌘OT
Time Profiler
Critic Browser
Process Browser
File Browser
Recent Messages
Change Sorter
Recover lost changes...
Emergency Evaluator
Screenshot

Standard development tools





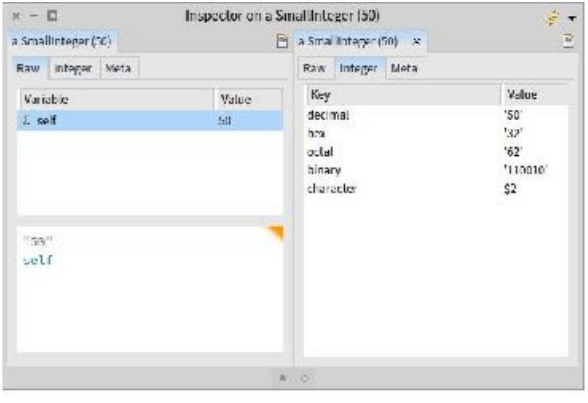


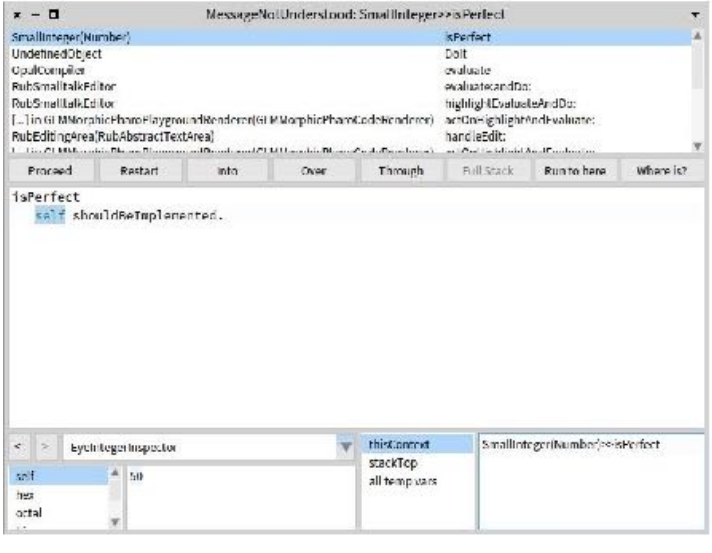


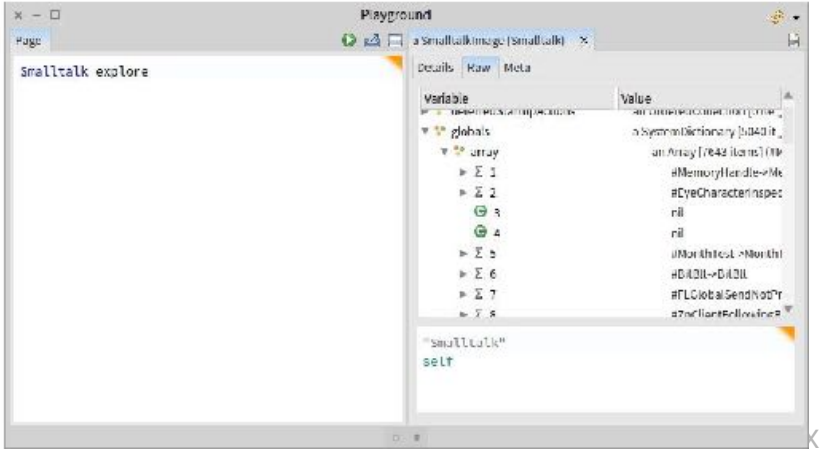
Here we see (clockwise from top left)

- the Playground, for executing arbitrary Smalltalk code
- the System Browser, for browsing (top) packages, classes, interfaces (or “protocols”), methods, and (bottom) source code
- the Monticello Browser, for connecting to a Monticello version repository
- the Test Runner, for (you guessed it!) running tests, and
- the Transcript, for displaying console messages.

Debuggers, Inspectors, Explorers



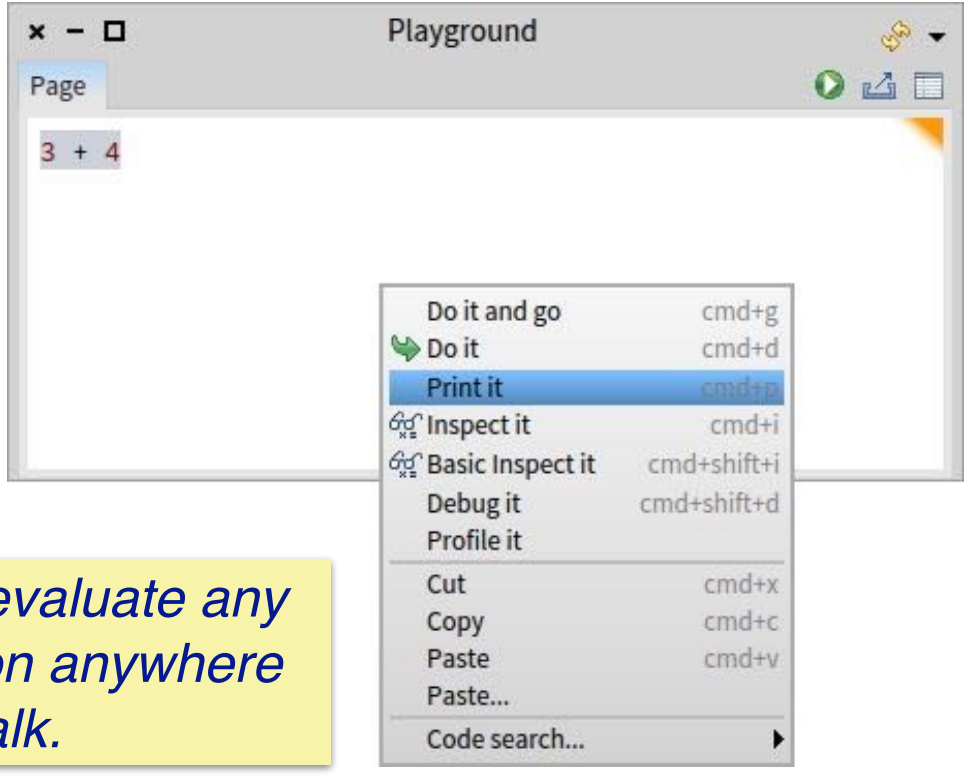




Here we see

- a Playground, where we inspect the expression
  `50 isPerfect`
- an object Inspector on the result of evaluating this expression
- another Playground where we evaluate and inspect Smalltalk explore in a separate tab
- a Debugger window showing us that the method `isPerfect` is just a stub to be implemented.

Do it, Print it, …



You can evaluate any expression anywhere in Smalltalk.

Here we have selected the text `3+4` in the Playground and clicked the “yellow” button to display a menu of operations we can perform.

- **do it** will evaluate it and discard the result
- **print it** will display the result
- **inspect it** will open an Inspector on the result

and so on

Roadmap

- The origins of Smalltalk
- What is Smalltalk?
- **Syntax in a nutshell**
- Seaside — web development with Smalltalk

Three kinds of messages

- *Unary messages*

```
5 factorial
Transcript cr
```

- *Binary messages*

```
3 + 4
```

- *Keyword messages*

```
3 raisedTo: 10 modulo: 5

Transcript show: 'hello world'
```

Smalltalk has a very simple syntax. There are just three kinds of messages:

1. *Unary messages* consist of a single world sent to an object (the result of an expression). Here we send `factorial` to the object `5` and `cr` (carriage return) to the object `Transcript`. (Aside: upper-case variables are global in Smalltalk, usually class names. Transcript is one of the few globals that is not a class.)

2. *Binary messages* are operators composed of the characters `+`, `-`, `*`, `/`, `&`, `=`, `>`, `|`, `<`, `~`, and `@`.
   Here we send the message "`+ 4`" to the object `3`.

3. *Keyword messages* take multiple arguments. Here we send "`raisedTo: 10 modulo: 5`" to `3` and "`show: 'hello world'`" to `Transcript`.

Precedence

First unary, then binary, then keyword:

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

128

Same as:

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

Use parentheses to force order:

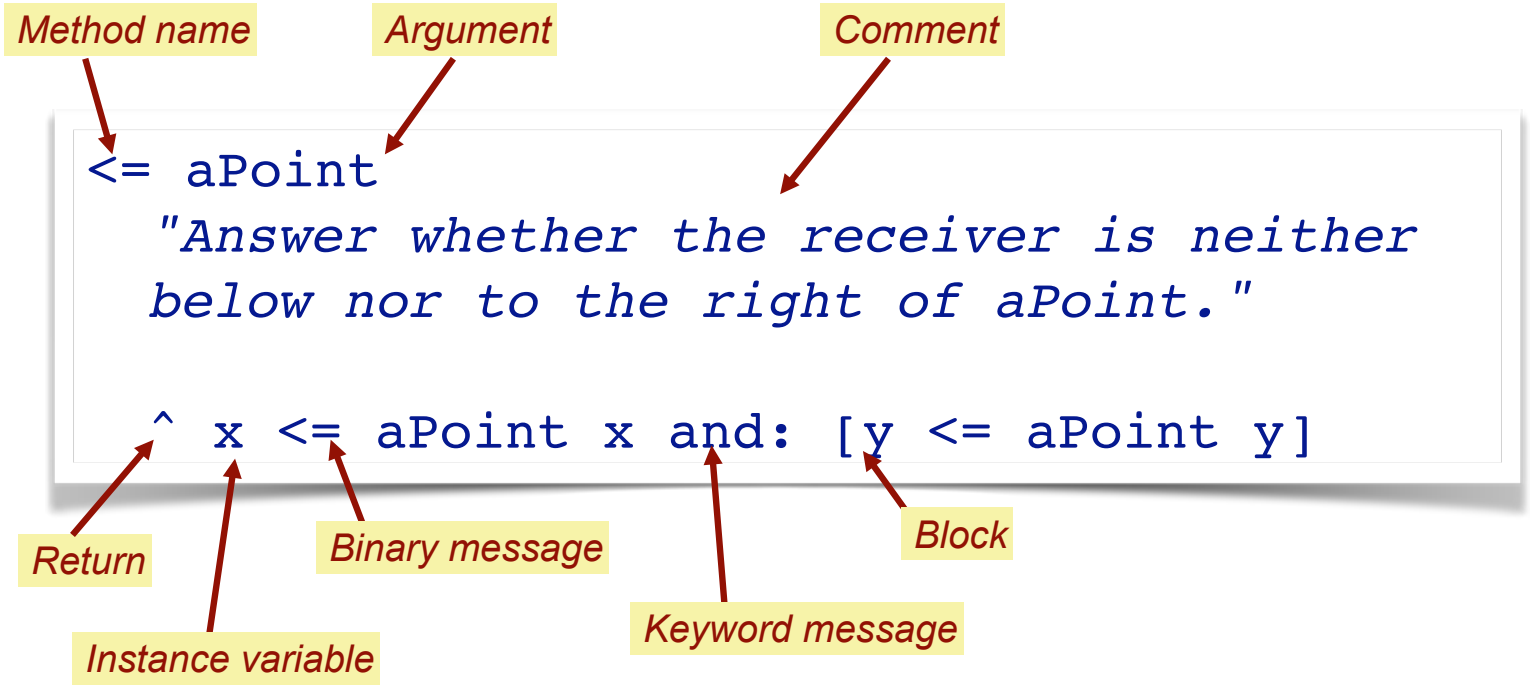
```
1 + 2 * 3
1 + (2 * 3)
```

9 (!)
7

The precedence rules for Smalltalk are exceedingly simple: unary messages are sent first, then binary, and finally keyword messages. Use parentheses to force a different order.

Note that there is no difference in precedence between binary operators.

A typical method in the class Point




```
(2@3) <= (5@6)
```

```
true
```

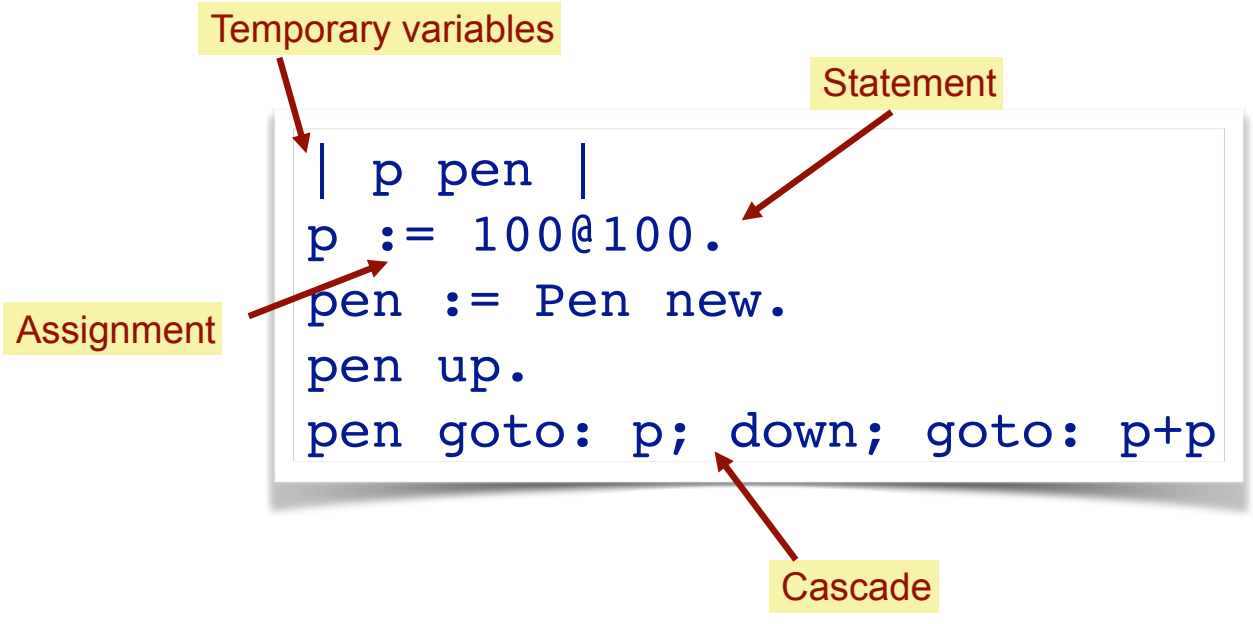
The slide shows the <= method of the `Point` class as it appears in the IDE.

The first line lists the method name and its formal parameters. In this case we are defining the method for the <= selector. (In Smalltalk, method names are called "*selectors*", because when a message is received, the selector is used to select the method to respond.)

Comments are enclosed in double quotation marks (*strings* are enclosed in single quotes).

The body of this method consists of a single expression. The caret (^) is a reserved symbol in Smalltalk and denotes a *return value*. A *block* is enclosed in square brackets and denotes an expression that may be evaluated. In this case, the Boolean `and:` method will only evaluate the block if its receiver (i.e., the subexpression to the left of the `and:`) evaluates to `true`.

Statements and cascades



This is a code snippet (not a method) that may be evaluated in the Playground.

Here we see that *statements* are expressions separated by periods (.).

Even though Smalltalk does not support type declarations, *local variables must still be declared*, appearing within or-bars (`|`).

A variable is bound to a value using the *assignment* operator (`:=`).

Smalltalk supports a special syntax, called a *cascade*, to send multiple messages to the same receiver. Messages in a cascade are separated by semi-colons (`;`). In this case we send the messages "`goto: p`", "`down`", and finally "`goto: p+p`" to the receiver `p`. (This draws a line from the `Point` `100@100` to `200@200`.)

Note that `100@100` looks like special syntax for `Point` objects, but it is really just a Factory method of the `Number` class, which creates a new `Point` instance.

Literals and constants

Strings & Characters	`'hello'  $a`
Numbers	`1  3.14159`
Symbols	`#yadayada`
Arrays	`#(1 2 3)`
Pseudo-variables	`self super`
Constants	`true false`

Everything is an object in Smalltalk, including these literal and constant values.

Strings are just special kinds of ordered collections holding character values.

Smalltalk supports various kind of numbers, and also supports radix notation for numbers in different bases.

Symbols behave much like strings, but are guaranteed to be globally unique. They always start with a hash (#).

In addition to `self`, `super`, `true` and `false`, there are only two further reserved names in Smalltalk: `nil` and `thisContext`. (The latter is only needed for meta-programming!)

Variables

- *Local variables* are delimited by | *var* |
  Block variables 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
```

```
(OrderedCollection with: 10 with: 5) collect: [:each| each factorial ]
```

```
an OrderedCollection(3628800 120)
```

NB: Since source code for methods in the IDE does not show the class of the method, it is a common convention in documentation to add the missing class name, followed by two greater-than signs (>>), as in this example.

This example serves mainly to show that blocks can take arguments. The arguments are after the opening left square bracket, and each is preceded by a colon (`:`).

The block:

```
[:each| each factorial ]
```

takes its arguments from the receiver of `collect:`, the collection holding `10` and `5`.

Control Structures

- *Every control structure is realized by message sends*

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

```
4 timesRepeat: [Beeper beep]
```

There are no built-in control constructs in Smalltalk. *Everything happens by sending messages!*

Even a simple if statement is achieved by sending a message to a boolean expression, which will then evaluate the block argument only if it boolean is true.

Here we see that the `max:` method is implemented by sending `ifTrue:ifFalse:` to the Boolean expression `self<aNumber`. The `ifTrue:ifFalse:` method is itself defined in the Boolean classes `True` and `False`.

(Try to imagine how it would be implemented, and then check in the image to see how it is done.)

Creating objects

- *Class methods*

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

- *Factory methods*

```
1@2
1/2
```

a Point

a Fraction

Ultimately all objects (aside from literals) are created by sending the message `new` to a class. (The message `new:` is used to create arrays of a given length.) Further constructors may be defined as convenience methods on classes, for example,

```
Array with: 1 with: 2
```

will create an `Array` of length 2 using `new:`, and then initialize it with the two arguments.

Other instance creation methods may be defined on the classes of arguments used to create the objects. For example, to create a `Fraction`, we send the message `/` to an `Integer`, with the numerator as its argument. This method will then actually create a new `Fraction` for us.

Creating classes

- Send a message to a class (!)

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

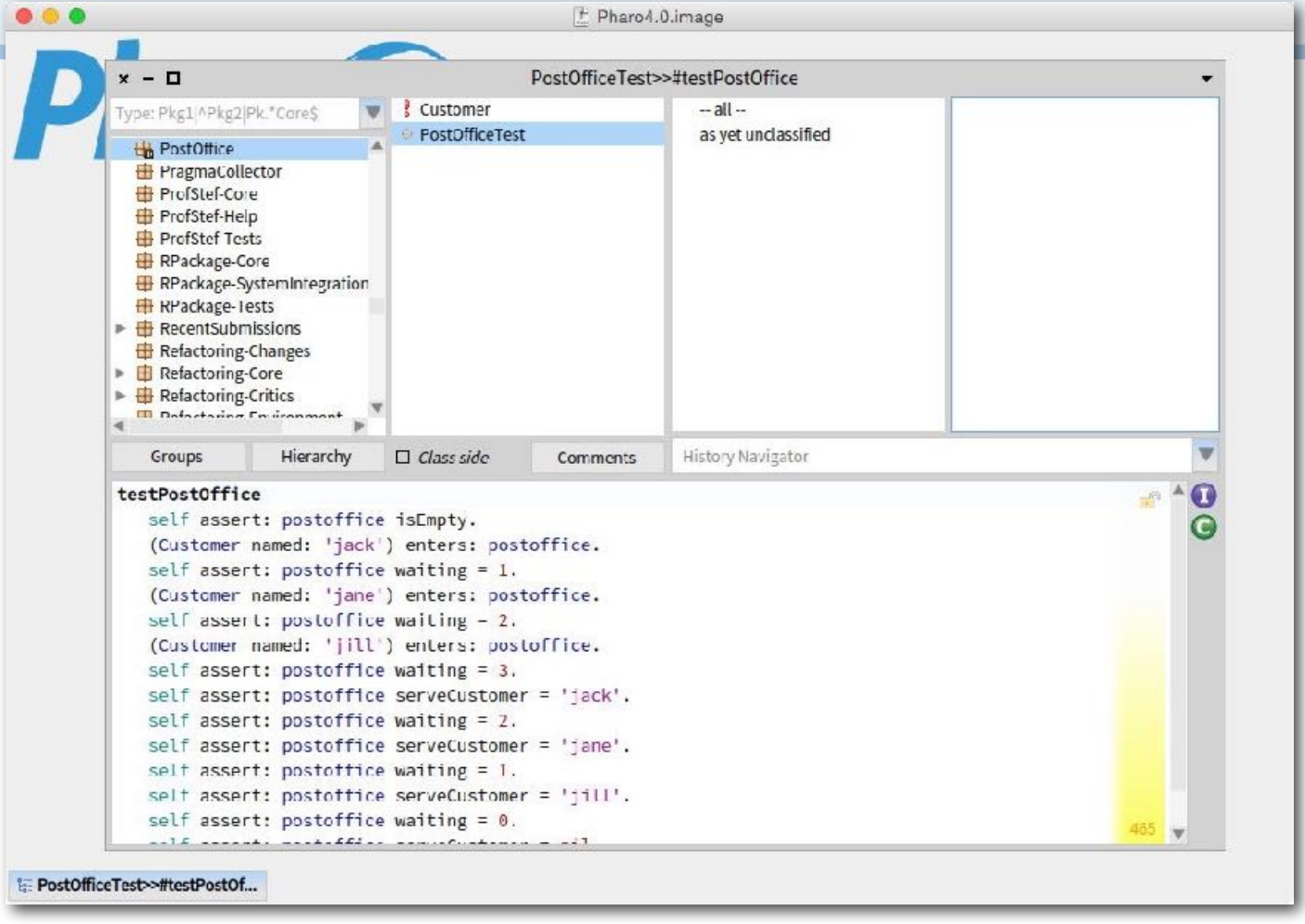
Everything is an object, ergo *classes are objects too*!

To create a new class, you must send a message to an existing class, asking it to create (or redefine) a subclass.

Since the class to be created probably does not yet exist, its name is not defined globally, so we must pass in the name as a symbol (here `#Complex`).

We can also provide the names of its instance variables (or we can update this later). Please ignore classVariableNames and PoolDictionaries — they are almost never needed. The "*category*" is the name of a related group of classes (something like a poor man's package).

Demo: Defining classes and methods



This demo script can also be found in the same github repo listed earlier.

Here we apply test-driven development to simulate a Post Office serving customers.

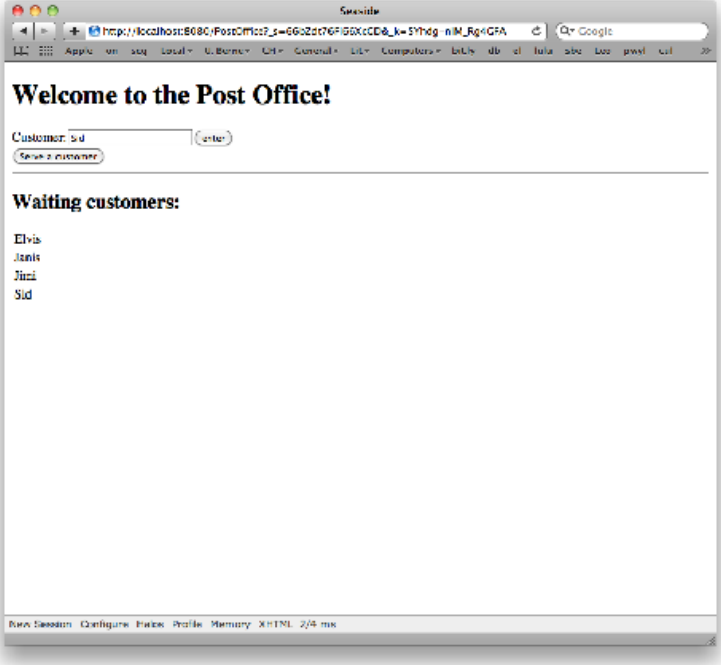
Roadmap

- The origins of Smalltalk
- What is Smalltalk?
- Syntax in a nutshell
- **Seaside — web development with Smalltalk**

Seaside — a Smalltalk web development platform

Seaside is a very successful web development platform built on top of Pharo Smalltalk. Like Pharo, it supports live programming.

Demo: PostOffice in Seaside



The demo shows how we can quickly develop a simple web interface to the Post Office simulation.

What you should know!

1. *What are the key differences between Smalltalk, C++ and Java?*
2. *What is at the root of the Smalltalk class hierarchy?*
3. *What kinds of messages can one send to objects?*
4. *What is a cascade?*
5. *Why does* `1+2/3 = 1` *in Smalltalk?*
6. *How are control structures realized?*
7. *How is a new class created?*
8. *What are categories for?*
9. *What are Factory methods? When are they useful?*

1)

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

The most important difference between Smalltalk, Java and C++, is that Smalltalk supports "live programming". Whereas in Java and C++ you must first write source code and compile it before you run anything, in Smalltalk you are always programming in a live environment. You incrementally add classes and compile methods within a running system.

As a consequence, Smalltalk has to be fully reflective, allowing you to reify ("turn in objects") all aspects of the system, and change them at run time. The only thing you cannot change from within Smalltalk is the virtual machine.

2) Object (or PrototypeObject)

3)

The only way to make anything happen is by sending messages. To ask "what can I do with this object?" is the same as asking "what messages does it understand?"

The terminology of "message sending" is perhaps unfortunate, as those new to Smalltalk often assume it has something to do with network communication, but one should understand it as a metaphor: you do not "call an operation" of an object, but you politely ask it to do something by sending it a request (a "*message*"). The object then decides how to respond by checking to see if its class has a "*method*" for handling this request. If it does, it performs the method. If not, it asks its superclass if it has such a method, and so on. If this search fails, the object does not understand the message (but let's not get into that now!).

4)

Smalltalk supports a special syntax, called a *cascade*, to send multiple messages to the same receiver. Messages in a cascade are separated by semi-colons (`;`). In this case we send the messages "`goto: p`", "`down`", and finally "`goto: p+p`" to the receiver `p`. (This draws a line from the `Point 100@100` to `200@200`.)

5) Unary first, then binary, then keyword-messages

6)

> *Every control structure is realized by message sends*

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

```
4 timesRepeat: [Beeper beep]
```

There are no built-in control constructs in Smalltalk. *Everything happens by sending messages!*

Even a simple if statement is achieved by sending a message to a boolean expression, which will then evaluate the block argument only if it boolean is true.

Here we see that the `max:` method is implemented by sending `ifTrue:ifFalse:` to the Boolean expression `self<aNumber`. The `ifTrue:ifFalse:` method is itself defined in the Boolean classes `True` and `False`.

(Try to imagine how it would be implemented, and then check in the image to see how it is done.)

7)

Everything is an object, ergo *classes are objects too*!

To create a new class, you must send a message to an existing class, asking it to create (or redefine) a subclass.

Since the class to be created probably does not yet exist, its name is not defined globally, so we must pass in the name as a symbol (here `#Complex`).

8)

~~PoolDictionaries — they are almost never needed~~. The "*category*" is the name of a related group of classes (something like a poor man's package).

g) When an object is created through a method instead through it's constructor.

- It's used when a class can't (or shouldn't) know the objects which are generated from that class. Or when subclasses should decide which objects are generated.

Can you answer these questions?

- *Which is faster, a program written in Smalltalk, C++ or Java?*
- *Which is faster to develop & debug, a program written in Smalltalk, C++ or Java?*
- *How are Booleans implemented?*
- *Is a comment an Object? How would you check this?*
- *What is the equivalent of a static method in Smalltalk?*
- *How do you make methods private in Smalltalk?*
- *What is the difference between = and ==?*
- *If classes are objects too, what classes are they instances of?*