Smalltalk

Based on Lecture
Prof. O. Nierstrasz
and
Pharo Online ProfStef
Tutorial
History
The History (Xeroc PARC Internal)

- **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
Dynabook Mockup

- At a time usually computers looked like THIS!
Alto: a Machine to Run Smalltalk

- Developed at the research center of Xerox in 1973
- Steve Jobs has visited Xerox PARC and has been fascinated by GUI
  https://de.wikipedia.org/wiki/Xerox_PARC
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
August 1981 Byte Magazine Smalltalk Issue

- Byte Magazine published a special issue in August 1981 that was completely devoted to the Smalltalk-80 Language and System. A key 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 was done by Robert Tinney and is of a multi-colored balloon leaving the ivory tower of Xerox labs.
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**
  - 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
Influence Concepts

- Eclipse IDE
- JIT compilation
- Java, Self, JavaScript
- 1st Apple Prototype
- Overlapping Windows
- Bit Blitting
- Objects, Garbage Collection, Byte Code, etc…

Extreme programming
- Unit Testing
- 1st wiki community
- Refactoring Browser
- Design Patterns
- Garbage Collection
- Model-View-Controller
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
# Smalltalk vs. C++ vs. Java

<table>
<thead>
<tr>
<th></th>
<th>Smalltalk</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object model</strong></td>
<td>Pure</td>
<td>Hybrid</td>
<td>Hybrid</td>
</tr>
<tr>
<td><strong>Garbage collection</strong></td>
<td>Automatic</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td><strong>Inheritance</strong></td>
<td>Single</td>
<td>Multiple</td>
<td>Single</td>
</tr>
<tr>
<td><strong>Types</strong></td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Reflection</strong></td>
<td>Fully reflective</td>
<td>Introspection</td>
<td>Introspection</td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td>Semaphores, Monitors</td>
<td>Some libraries</td>
<td>Monitors</td>
</tr>
<tr>
<td><strong>Modules</strong></td>
<td>Categories, namespaces</td>
<td>Namespaces</td>
<td>Packages</td>
</tr>
</tbody>
</table>
Smalltalk Basics and Environment
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
Hello World

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

- Transcript is a kind of “standard output”
Environment

- Every Smalltalk system is full of objects.
  - There are windows, text, numbers, dates, colors, points and much more. You can interact with objects in a much more direct way than is possible with other programming languages.

- 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).

- SimpleButtonMorph explore.
The Smalltalk Browser & Editor

- Right click on the World
- Packages = object categorie
- Classes & instances
- Protocols, method category
- Methods, selectors
- Enter name to find it
- Information e.g. method implementation
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.
  - 'hello word' is an instance of String.
  - show: is a Message
  - 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.
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
Natural Language "Alike"

```
album play
album playTrack: 1
album playFromTrack: 5 to: 10
```

- object.
- message.
- message with argument.
Accept, Dolt, PrintIt and InspectIt

- There is a context menu for selected strings

- **Dolt**
  - Evaluate an expression

- **PrintIt**
  - Evaluate an expression and print the result

- **ExploreIt**
  - Open in Object Browser
Persistent Object Memory

- Smalltalk is language *and* environment
- Everything inside the image
  - persistent objects
  - fully reflective system
  - incremental compilation
Save & Load Single Classes

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

- load a class: in Word -> open Tools -> FileBrowser "Install"
The Language
Doing vs Printing

- DoIt
  - execute the expression

- PrintIt It's a
  - 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!

1000 factorial

4023872600709377354370243492300398571937486421074632543799910429385123986290205920442084869404
8047998861019719065863166687299480855859013238296699445909975042450408707375991882367277188732519779
509595952761208749754624970436014182780946469629105639388473788648733711918104582578364784990712
4766328898359557354325131853239586430755574091142624174743493457534286465766116677973966668820291207
379143853719588249801812686783873455973174613608537953452422158659320192809087829730843139284403281
2315586110369768013573042161687460967587134831202547858932767619133244842636131412508780208000261
6831510273418279777047846358681701643650241553691398281264810212309276124489635992870511469475419909
34222156663252708028133181661811553615863654698407608795760290950537616475847728421889679646244945
1676753534081989013855424879849599533191017235555660213945039973628075013783761530712761926849034
3526252000158883513473316117021039681759215109077880193393718114194545257223865541461062892187960223
8389714760885062676826714667469756291123408243920816015378088989396453262436761672179169089779
911903754013127462228998800519544444142820212187361745992642956817466283029557029902432415318161720
4658320367869061172601587835207515162842255402651708330422614379428693306169089796842589250129545327
16822645806652676699586562822782007578131918581788896522081643483484825993266043367660176996612831860
78838615027946595513115655203609399818061213855860030143569452722420634463179746059468257310379084
024433483486565724501440482288525247093519062092902313649327349975655139587205596542287974770411413446
962715422845623773875382304838656688976419273838149001407673104464602598994994022221275904339901886
0185656248506179970235619389701786004081188972991831102171229845901641921068884378121855646124960
798722980519296819373288642614657224738829911231250241866493543143907137428531926649875373218940694281
4341185201580141233448280150513969642901534830776746565909073152433272882869640620789864321139083506
217095005257937889817647228248757586675752432202075736056948982508796892162753848863396909
9598262809561214509944871701244516461260379029309120889068942028510640182154399457156805941872748989
09425474217358240106367740455974178516082923013535808184000969637252423056085590370624271243416909
00415369015933983835779739941097002775347200000000000000000000000000000000000000000000000000000000000000000000000000000000
Automatic Coercion

- On demand numeric types are coerced automatically

\[
1 + 2.3 \quad \rightarrow \quad 3.3
\]

\[
1 \text{ class} \quad \rightarrow \quad \text{SmallInteger}
\]

\[
1 \text{ class maxVal class} \quad \rightarrow \quad \text{SmallInteger}
\]

\[
(1 \text{ class maxVal} + 1) \text{ class} \quad \rightarrow \quad \text{LargePositiveInteger}
\]

\[
(1/3) + (2/3) \quad \rightarrow \quad 1
\]

\[
1000 \text{ factorial} / 999 \text{ factorial} \quad \rightarrow \quad 1000
\]

\[
2/3 + 1 \quad \rightarrow \quad (5/3)
\]

\[
(1/3) \text{ class.} \quad \rightarrow \quad \text{Fraction}
\]
Explicit Coercion

- Types can also be converted explicitly

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

- Object hierarchie of numerical types

```
Object
  |
  ▼
Magnitude
  |
  ▼
Number
  |
  ▼
Float  Fraction  Integer
      |
SmallInteger  LargePositiveInteger
      |
          LargeNegativeInteger
```
Basic Types: Characters

- A Character can be instantiated using $ operator:

**Examples**

$A. \ $A

$A \ \text{class.} \ \text{Character}

$B \ \text{charCode.} \ \ 66$

print all 256 characters of the ASCII extended set:

```plaintext
Character allByteCharacters.

'!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`'
abcdefghijklmnopqrstuvwxyz{|}~

£¤¥¦§¨©ª«¬–°±²³´µ¶·¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖрÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷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'
Strings

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

- Strings are mutable

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

#mac asString
12 printString
String with: $A
'can''t' at: 4
'hello', ' ', 'world'

'mac'
'12'
'A'
'
'hello world'
```
Basic Types Symbols, Comparison

- 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

- 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*
Comments

- Comments are simply enclosed 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

- 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

```smalltalk
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 of True and False
  - Optimized and inlined

- Lazy evaluation with and: and or:

  \[
  \text{false and: } [1/0] \quad \text{false}
  \]

  \[
  \text{false \& (1/0)} \quad \text{ZeroDivide error!}
  \]
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 Books): ←
Assignment

- 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

```plaintext
| p1  p2 |
p1 := 3@4.
p2 := p1.
p2 5@6
```
Global Variables

- Always capitalized (convention)
  - If unknown, Smalltalk will prompt you to create a new Global
  - Stored in the Smalltalk System Dictionary

- Avoid them!
Global Variables

- To remove a global variable:

  Smalltalk removeKey: #MyGlobal

- Some predefined global variables:

<table>
<thead>
<tr>
<th>Smalltalk</th>
<th>Classes &amp; Globals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undeclared</td>
<td>A PoolDictionary of undeclared variables accessible from the compiler</td>
</tr>
<tr>
<td>Transcript</td>
<td>System transcript</td>
</tr>
<tr>
<td>ScheduledControllers</td>
<td>Window controllers</td>
</tr>
<tr>
<td>Processor</td>
<td>A ProcessScheduler list of all processes</td>
</tr>
</tbody>
</table>
Variables

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

Diagram:

```
variable
  └── Shared variable
      ├── Class variable
      │    └── Global variable
      │          └── Pool variable
      └── private variable
          └── temporary variable
              └── method parameter
                      └── method temporary
                          └── block temporary
```

```
Six Pseudo-Variables

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>A reference to the UndefinedObject</td>
</tr>
<tr>
<td>true</td>
<td>Singleton instance of the class True</td>
</tr>
<tr>
<td>false</td>
<td>Singleton instance of the class False</td>
</tr>
</tbody>
</table>
| self     | Reference to this object  
           Method lookup starts from object’s class |
| super    | Reference to this object (!)  
           Method lookup starts from the superclass |
| thisContext | Reification of execution context |
Message Syntax: Unary Messages

- 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
ture & true.  true
ture & false.  false
10 @ 100.  (10@100)
10 <= 12.  true
'ab', 'cd'.  'abcd'
Date today < Date yesterday.  false
Keyword Messages are messages with arguments.

They have the following form:

\[
\text{anObject akey: anotherObject [ akey2: anotherObject2 ]}
\]

**Examples**

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

"The message is r:g:b: implemented on class Color."
Message Syntax: Execution Order

- 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

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

\[ 2 \times 10 + 2. \quad 22 \]

- 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 !.

\[ 2 + 2 \times 10 \]. \quad 40 \]
\[ 2 + (2 \times 10). \quad 22 \]
\[ 8 - 5 / 2. \quad (3/2) \]
\[ (8 - 5) / 2. \quad (3/2) \]
\[ 8 - (5 / 2). \quad (11/2) \]
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

- 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**

```
[Browser open].
```

- does not open a Browser because the block is not executed.

```
[Browser open] 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`.  

School of Engineering © O. Nierstrasz, K. Rege, ZHAW
Block Assignation

- 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'].

[Transcript window showing 'No : 3 is less than 10']
For Loops

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

Examples

1 \texttt{to: 100 do:}

\[
[:i \mid \text{Transcript show: i asString; cr}].
\]

1 \texttt{to: 100 by: 3 do:} [:i \mid \text{Transcript show: i asString; cr}].

100 \texttt{to: 0 by: -2 do:}

\[
[:i \mid \text{Transcript show: i asString; cr}].
\]
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

```plaintext
[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.

```smalltalk
#(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

MessagePublisher new

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

  publisher := MessagePublisher new

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

  MessagePublisher new publish
Instantiation

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

- After new the initial values can be set and methods can be invoked

  SimpleButtonMorph new
  label: 'Open Transcript';
  target: [Transcript open.];
  actionSelector: #value;
  openCenteredInWorld.

- 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'!

Number subclass: #Complex
    instanceVariableNames: 'real img'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ComplexNumbers'
Define new Classes via Browser

Beliebige Klasse Auswählen und KlassenNamen und category setzen

HelloMessage ctrl-s to save
Add Instance Message in new Msg Category

- Select the instance method to change implementation.
- Right click to rename.
- Select the instance method again.
- Edit the implementation.
- Press ctrl-s to save.

Execute 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


School of Engineering © O. Nierstrasz, K. Rege, ZHAW
Message and Variables

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

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

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

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

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

```plaintext
max: aNumber
   ^ self < aNumber
      ifTrue: [aNumber]
      ifFalse: [self]
\[1\ max: 2\]
```

■ Methods *always* return a value
  ■ By default, methods return `self`
Return Value of Messages Examples

- use caret (^) to specify a value from a method or a block

**Example**

```smalltalk
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
```

```smalltalk
Complex>>+ aComplex

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

^ nComplex
```
Object Instance Methods

- Every Object supports following messages (and more)

- class
  returns the receiver class

- isKindOf: aClass
  whether aClass is a superclass of the receiver

- respondsToSelector: 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.
Reflection

■ You can inspect *and change* the system at runtime.
SimpleButtonMorph definition.

■ just its comment:
SimpleButtonMorph comment.

■ Here's all the methods implement:
SimpleButtonMorph selectors.

■ Create and add a new *doSth* method and call it
SimpleButtonMorph class
compile: 'doSth Transcript open'.
SimpleButtonMorph doSth.

■ Remove It
SimpleButtonMorph class removeSelector: #doSth.
Examples
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 class `Node`)
    
    `Node class>>withName: aSymbol`

- `aSomething` is an instance of the class `Something`
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
Slow Fibonacci

- Recursive Method

\[
Fibs>>at: \text{anIndex}
\begin{align*}
\text{anIndex} = 1 \text{ ifTrue: } [ ^1 ] \\
\text{anIndex} = 2 \text{ ifTrue: } [ ^1 ] \\
^1 (\text{self at: anIndex - 1}) + (\text{self at: anIndex - 2})
\end{align*}
\]

\[\text{Fibs new at: 35} \quad 9227465\]

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
Caching Fibonacci

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

```smalltalk
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!
A Simple Drawing Canvas

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

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

drawBlock: aBlock
drawBlock := aBlock.
self changed.

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

use e.g. in Workspace
m := CanvasMorph new openInWorld.
m delete.

School of Engineering © O. Nierstrasz, K. Rege, ZHAW
Drawing Samples

■ some sample Canvas Drawing

```smalltalk
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

```smalltalk
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.
    ]
]
```
Questions
License

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

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.