Today’s Schedule

A. Course Overview
B. Introduction to Reflection and Metaprogramming
C. Introduction to Smalltalk
D. Principles of Reflection
Introduction to Smalltalk*  

* slides created with the help of Johan Brichau and Kris Gybels
Smalltalk in a nutshell (1)

- Pure object-oriented language:
  - Everything is an object
  - Objects communicate only by message passing

- Class-based language:
  - Every object is an instance of a class
  - All classes have a (single) parent class
  - Object is the root class

- Uniform & simple syntax
  - Forces one to think object-oriented
  - Syntax fits on a single sheet of paper
  - Statements read almost like English phrases
  - Originally designed as a learning environment for children

- Dynamically typed

- Fully reflective language
  - IDE entirely written (and modifiable) in Smalltalk itself
Smalltalk in a nutshell (2)

- Platform independent
  - A runtime environment based on VM technology
- Encourages exploratory programming
- Memory management (garbage collection)
- A large set of reusable classes
  - basic data structures, GUI classes, DB access, internet, graphics, ...
- Powerful development tools
  - browsers, GUI builders, inspectors, change management tools, crash recovery tools, project management tools, unit testing, refactoring, ...
- Team working environment
  - releasing, versioning, deploying
- Source of inspiration to many other languages and environments
  - environment, collections, debugger, reflective features and many development tools
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
  - IMAGE2.IM
  - IMAGE2.CHA
  - One per user
  - A byte-code interpreter: the virtual machine interpretes the image
  - Standard SOURCES
    - Shared by everybody

- **Image = bytecode**
- **Sources and changes = code (text)**
Smalltalk is a “pure” OO language

Objects

- a Point
  - x
  - y

Methods

- add:
  - a SmallInteger
    - add: extra
      - x := x + extra.
      - y := y + extra.

Classes

- a SmallInteger
  - 5
- a SmallInteger
  - 10
- a SmallInteger
  - 15
- Point
  - x, y
  - add: extra
  - x := x + extra.
  - y := y + extra.
Everything is an object

Objects have state

- **a Person**
  - name
  - address

- **an Address**
  - street
  - nr

- **a ByteString**
  - ‘Kim Mens’

- **a ByteString**
  - ‘Pl. Sainte Barbe’

- **a SmallInteger**
  - 2
Everything is an object

- Workspaces, windows, text editors, scrollbars, ... are all objects.
- ‘hello word’ is an object: it is an instance of String.
- #show: is a Symbol that is also an object.
- Numbers are instances of classes: 1 is instance of Integer
- Booleans are instances of their class: true is instance of class True
- The parser, compiler and garbage collector are objects.
- Even classes are objects themselves

- 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.

- Smalltalk is a fully reflective language
# Basic Smalltalk Expressions

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td>1234, $a, #Johan, ‘Kim’, #(#a #b)</td>
</tr>
<tr>
<td><strong>Variable Reference</strong></td>
<td>someObject, a, GlobalVariable</td>
</tr>
<tr>
<td><strong>Message Expressions</strong></td>
<td>someObject doSomethingWith: 1234, 1 + 2, ‘Kim’, ‘Mens’, #(a b) first</td>
</tr>
<tr>
<td><strong>Return Expressions</strong></td>
<td>^someObject print: ‘Johan’</td>
</tr>
<tr>
<td><strong>Block Expressions</strong></td>
<td>[:x :y</td>
</tr>
</tbody>
</table>
Literal Expressions

- 'Pleinlaan' = a ByteString
- 42 = a SmallInteger
- #Brussels = a ByteSymbol
- $a = a Character

- #(1 #a $c) = an Array
- = a SmallInteger
  | 1
  2
  3

- = a ByteSymbol
  | #a

- = a Character
  | $c
Exercise session “Introduction to Smalltalk”

Exercise 1

- predict the result of some basic Smalltalk expressions and evaluate them
Objects can be sent messages ...

... and reply by executing a method

```
add: extra
x := x + extra.
y := y + extra.
```

```
+----+----+
|    |    |
|    |    |
+----+----+
```

```
+----+----+
|    |    |
+----+----+
```

```
+----+----+
|    |    |
+----+----+
```

```
+----+----+
|    |    |
+----+----+
```

Objects communicate using message sending

- A **message** is a request to an object to do something
- A **method** is piece of source code belonging to a class

**Transcript** show: ‘hello world’

The above expression is a message

- **Transcript** is a global variable containing a Stream object that represents a “console” window.
- **show:** ‘hello world’ is a message sent to a Stream object
  - The stream object (value of Transcript) is the **receiver** of the message
  - #show: is the **selector** of the message
  - the string ‘hello world’ is the (actual) **argument** of the message
An object executes a method when it receives a message.

- Object 'hello world' receives a request to reverse itself
- Method reverse is defined on the class String
- Executing this method returns a new string 'dlrow olleh'

Transcript show: 'hello world'

- Object Transcript receives a request to show a string and results in the string being printed on the transcript window.

Message passing or sending a message is similar to

- invoking a method in Java or C++
- calling a procedure in procedural languages
- applying a function in functional languages
Three kinds of messages

```
\begin{align*}
&\text{a SomeObject} \\
&\text{someVar} \\
&\text{printYourself} \\
&+ \\
&\text{add: with:} \\
&\text{a Number} \\
&\text{42} \\
&\text{a Number} \\
&\text{100}
\end{align*}
```
Unary messages

selector is alphabetic name
no arguments

selector is printString

printString → an Assistant
name

‘Johan Brichau’

= 

a ByteString

‘Brichau J.’
Unary messages

42 cos

= a SmallInteger

42

= a Float

-0.399985
Binary messages

selector is one or two special characters exactly one argument (and one receiver)
Selector is sequence of names (keywords) ending in colon
One argument for each keyword

**Keyword messages**

Selector is sequence of names (keywords) ending in colon
One argument for each keyword

```plaintext
a SomeObject
someVar

add: with: andPrint:
```

```plaintext
a SmallInteger
42
```

```plaintext
a SmallInteger
100
```

```plaintext
a ByteString
Boo'
```
Exercise session “Introduction to Smalltalk”

Exercises 2-5

- evaluate some message expressions and indicate the receiver, selector, arguments, message and result
Unary message expressions

\[ 42 \cos \sin = \cos 42 \text{ SmallInteger} \]

\[ \cos 42 \text{ SmallInteger} = 0.399985 \]

\[ \sin 42 \text{ Float} = -0.399985 \]

\[ = \text{ a Float} -0.389405 \]
Binary message expressions

\[
\begin{align*}
1 + 2 \cos & = 1 + (2 \cos) \\
1 + 2 \cos - 3 & = (1 + (2 \cos)) - 3 \\
1 + 2 \cos + 3 & = (1 + (2 \cos)) + 3 \\
1 + 2 \times 3 + 4 & = ((1 + 2) \times 3) + 4
\end{align*}
\]
Keyword message expressions

Time readFrom: (ReadStream on: ‘1:30:00 pm’)

=*

Time readFrom: ReadStream on: ‘1:30:00 pm’
Precedence rules

- Unary message
  - `>>` binary message
  - `>>` keyword message

- Evaluation from **left to right**

- There are **no** specific precedence rules for the arithmetic or logic operators

2 negated raisedTo: 3 + 2 * 5 is equivalent with

(2 negated) raisedTo: ((3 + 2) * 5)
Exercise session “Introduction to Smalltalk”
Exercises 6-8
  on message precedence rules
More exercises ...

- Exercise session “Introduction to Smalltalk”
- Exercises 9-15
  - on symbols, strings and comparisons
Blocks are “delayed” expressions

Blocks can be sent messages to evaluate the body of the block

[ :foo :bar | foo + bar ] value: 4 value: 2 = a SmallInteger 6
Block expressions

[ Transcript show: 'Boo!' ] value

ablock := [ :someName | someName asUppercase ].
ablock value: 'Kim'

[ :a :b | a + b ] valueWithArguments: (Array with: 1 with: 2)
Exercise session “Introduction to Smalltalk”
Exercises 16-19
  on Smalltalk code blocks
Control structures

- In Java different predefined syntactic constructs like >>, if, for, while, ... are hard-coded into the language grammar.

- In Smalltalk, they are just defined on objects, e.g. :

  ```
  (>>) bitShift: is just a message sent to numbers
  10 bitShift: 2
  
  (if) ifTrue: is just messages sent to a boolean
  (1 > x) ifTrue: [ ... ]
  
  (for) do:, to:do: are just messages to collections or numbers
  #(a b c d) do: [:each | Transcript show: each ; cr]
  1 to: 10 do: [:each | Transcript show: each printString ]
  ```

- Advantages :
  - Language is extensible (very easy to add “new syntax”)
  - Simplicity & orthogonality (uniform language design)
Blocks example: if then else

```plaintext
ifTrue:    ifFalse:
  a BlockClosure
  a BlockClosure
  value
  a True
```
Blocks example: if then else

ifTrue: ifFalse: → a False

a BlockClosure

value

a BlockClosure
true ifTrue: [ Transcript show: 'yes' ] ifFalse: [ Transcript show: 'no' ]

(a < b ) ifTrue: [ Transcript show: 'a smaller' ]
  ifFalse: [ Transcript show: 'a not smaller' ]

Transcript show: ((a < b) ifTrue: [ 'a' ] ifFalse: ['a not']) , ' smaller'
Hands on ...

- Visit the Boolean classes True and False
  - try to understand how the logic operators and, not, or are implemented
  - try to understand how ifTrue:ifFalse: is implemented

- Exercise session “Introduction to Smalltalk”
- Exercises 20-24
  - on conditionals and iteration with Smalltalk blocks
A method that illustrates every part of Smalltalk method syntax except primitives.

```smalltalk
uselessMethod: x
    \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
```
This (useless) method illustrates the entire Smalltalk syntax

- messages
  - unary, binary, and key word messages
  - cascaded messages

- variables
  - arguments and temporaries (local variables): no block temporaries
  - global variables
  - accessing and assigning a variable
  - pseudo variables (true, false, nil, self, super)

- instructions
  - sequence of instructions (.)
  - return statement (^)

- blocks (both zero argument and one argument block)
- literals (array, character, symbol, string, integer, float)
Smalltalk Syntax (1)

- comment
- character
- string
- symbol
- array
- integer (r = radix)
- real (e = exponent)
- fraction
- boolean
- point
- VM primitive call
- parenthesis (precedence)

“a comment”
$c $t $e $r $# $@
’a string’ ‘lulu’ ‘l”idiot’
#mac #+
#(1 2 3 (1 3) $a 4)
1, 2r101
1.5, 6.03e-34, 4, 2.4e7
1/33
true, false
10@120
<primitive: ...>
( ... )

2r101 = 5 in binary representation
Smalltalk Syntax (2)

- assignment
- a block
- local variable (method / block)
- block variable
- unary message
- binary message
- keyword message
- cascading of messages
  = sending several messages to the same instance
- sequence of instructions
- return statement

```
var := aValue
[:var | |tmp| expr...]
|tmp|
:var
receiver selector
receiver selector arg
receiver kw1: arg1 kw2: arg2
message ; selector; ...

message . message
^ message
```
## Smalltalk versus Java

<table>
<thead>
<tr>
<th></th>
<th>Smalltalk</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>Control structures</td>
<td>Defined by the programmer</td>
<td>Predefined</td>
</tr>
<tr>
<td>Typing</td>
<td>Dynamic</td>
<td>Static (primitive types, type casts)</td>
</tr>
<tr>
<td>Reflection</td>
<td>Full reflection</td>
<td>Mainly introspection</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Virtual machine</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Processes</td>
<td>Threads</td>
</tr>
<tr>
<td>Visibility of variables</td>
<td>Protected</td>
<td>Public, protected, private</td>
</tr>
<tr>
<td>Visibility of methods</td>
<td>Public</td>
<td>Public, protected, private</td>
</tr>
<tr>
<td>Interfaces</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Smalltalk versus Java

- Java and C++ have a (too) complex syntax
  - The constructs for expressing conditionnals, iteration, arithmetic and so on are predefined in the syntax
  - Conditionnals
    - if (BooleanExpression) { /* do something */ }
    - if (BooleanExpression) { /* do something */ }
      else { /* something else */ }
    - case ...
  - Iteration
    - for (i=1; i<length; i++) { /* do something */ }
    - while (BooleanExpression) { /* do something */ }
  - Arithmetic
    - 2 + 4
    - 3 * 7
    - 10 << 2 (bitshift)
Smalltalk has a much more simple and uniform syntax

- Everything is a message, even
  - Conditionnals
  - Boolean operators
  - Arithmetic operators
  - Iteration

- The language is extensible
  - Define your own language constructs
Hands on ...

- Finish exercise session “Introduction to Smalltalk”
- Exercises 25–27
  - creating a Smalltalk class
creativecommons.org/licenses/by-nc-sa/3.0