Smalltalk

Johan Brichau (LIFL/PROG) & Roel Wuyts (ULB)

Dynamic Languages Day, February 13, 2006
Vrije Universiteit Brussel
Part 1
Smalltalk Basics
Smalltalk at a glance

- Pure object-oriented language:
  - everything is an object (Integer, Class, Compiler, ...)
  - only message sends (almost no syntax).
- Single Inheritance
- Meta-programming and (full) reflection
- Dynamically typed
Smalltalk at a glance (ctd)

• Visibility
  • instance variables are private to the object,
  • methods are public
• Call by reference (e.g. everything is a pointer)
• Garbage collector
• Virtual machine
• Incremental compilation
Smalltalk syntax

- Three kinds of message sends:
  - unary
  - binary
  - keyword

- Pseudo variables
  - self
  - super
  - true
  - false
  - nil
  - thiscontext

- Examples:
  - 'Smalltalk' printString
  - 2@5
  - 1+3
  - 5/9
  - dict at: #hhgtg put: 42
Syntax

- comment
- character
- string
- symbol
- integer
- real number
- fraction
- literal array

```
"a comment"
$a  $#
'Smalltalk' 't''s'
#mac  #+
42  2r101
1.5  6.03e-34
1/33
#(1 2 3 (1 3) $a 4)
```
Syntax

- assignment
  - `var := 5`
- local variable
  - `| var counter |
- block variable
  - `:var`
- separator
  - `expr1 . expr2`
- return
  - `^42`

- and finally... block
  - `[ ... ]`
Syntax

• Everything else are messages sent to objects!

(5 > 4) ifTrue: [^Set new]
x bitShift: 2
1 to: 10 do: ...

• Advantages
  • minimal parsing
  • simple parse tree; ideal for OO research
  • language is extensible
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
   (EmployeeList new scanFile:
      (Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
   Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
(EmployeeListView new scanFile:
(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
(EmployeeListView new scanFile:
(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:

(EmployeeList new scanFile:
  (Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
  Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:

(EmployeeList new scanFile:

(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:

Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:

(EmployeeList new scanFile:

(Filename named: ’employee.dat’) readStream)

aRectangle := (Point setX: 25 setY: 50) corner:

Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
(EmployeeList new scanFile:
(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:

(EmployeeList new scanFile:

(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:

Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
(EmployeeList new scanFile:
(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
Cursor currentCursor mousePoint + offset
Message Precedence

( ), unary, binary, keyword and left to right

EmployeeListView openOn:
(EmployeeList new scanFile:
(Filename named: 'employee.dat') readStream)

aRectangle := (Point setX: 25 setY: 50) corner:
Cursor currentCursor mousePoint + offset
Delayed Evaluation: Blocks

- Code inside a block is not directly evaluated.
- Only when the block receives
  - value
  - value:value:
  - ...

Delayed Evaluation Example:

```smalltalk
| count countBlock |
count := 0.
countBlock := [count := count+ 1].
countBlock value.
countBlock value.
^count

| index sumBlock |
index := 0.
sumBlock := [:x :y | index := x+y].
sumBlock value:5 value: 19.
^index
```
Block Expression

- Lexically scoped
- Executed in scope of definition context
- Re-entrant
- Full closures
Some block examples

| aBlock |
aBlock := [:val | val > 0
  ifTrue: [val + (aBlock value: val - 1)]
  ifFalse: [0]].
aBlock value: 6

#(1 2 3 4 5 6) do: [:item |
  Transcript show: item asString ]

#(1 2 3 4 5 6) collect: [ :item | item + 1 ]

#(1 2 3 4 5 6) select: [ :item | item < 3 ]
Core classes

- Let’s have a look at
  - booleans
  - conditionals & loops
  - collections

- All of these are part of the class library
  - not hardcoded in the language!
  - implementation is available in environment

- learn by example
Booleans

(Random new next * 10) rounded >= 5
ifTrue: [Transcript show: 'Oeh']
ifFalse: [Transcript show: 'Aah']

4 > 2 | (9 > 7) ifFalse: [ ... ]

(4 > 2 and: [1/0 > 8]) ifFalse: [ ... ]

(2 > 4) not ifTrue: [ ... ]
Boolean hierarchy

ifTrue: trueBlock ifFalse: falseBlock
  ^falseBlock value

or: alternativeBlock
  ^alternativeBlock value

&: alternativeObject
  ^self

not
  ^true
true and false

- true and false are the sole instances of respectively the class True and False
- Singleton design pattern
Loops

| counter max |
max := 10.
number := 1.
[number <= max] whileTrue: [
  Transcript show: number.
  number := number + 1
]

1 to: 10 by: 3 do: [:number |
  Transcript show: number
]
Conditional & Loop classes

Number>>to: stop by: step do: aBlock
(Interval from: self to: stop by: step)
do: aBlock

BlockClosure>>whileTrue: aBlock
^self value
ifTrue:
[aBlock value.
[self value] whileTrue: [aBlock value]]
<table>
<thead>
<tr>
<th>anArray aSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>aSet := aArray asSet.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>weekdays</th>
</tr>
</thead>
<tbody>
<tr>
<td>weekdays := #(mon tue wed thur fri).</td>
</tr>
<tr>
<td>weekdays</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>do: [:day</td>
</tr>
<tr>
<td>separatedBy: [Transcript space]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>str := 'mysettings.txt' asFileName writeStream.</td>
</tr>
<tr>
<td>[ str nextPutAll: 'some text' ] ensure: [str close]</td>
</tr>
</tbody>
</table>
Collection Hierarchy (part)

- **Collection**
  - `add:`
  - `do:`
  - `remove:ifAbsent:`
  - `size`
  - `collect:`
  - `select:`
  - `detect:`
  - `includes:`
  - ...

- **SequencableCollection**
  - `first`
  - `last`
  - `indexOf:`

- **SortedCollection**

- **Set**

- **OrderedCollection**

- **ArrayedCollection**

- **Array**

- **CharacterArray**
  - `asText`
  - `string`
  - `<`
  - `>`
  - `match:`
  - `findString:startingAt:`

- **Stream**
  - `contents`
  - `flush`
  - `next`
  - `nextPut:`

- **Set**

- **String**

- **Symbol**
Meta programming in ST

- Everything is an object
- Class is an object itself
- So you can pass it around, store it, compare it, inspect it, send messages to it, ...
- Every object has a class
One immediate side-effect

- Constructors are not needed
- Class methods are used instead
- Just methods
- Can be inherited, extended, ...

Person new

Array with: 1 with: 2
Reflection

- Smalltalk program can, at runtime
  - ask information about itself (introspection)
  - change itself (intercession)
- More about this in the hands-on part...
Part 2
Hands-on Demo
**Example: Scaffolding Pattern**

- We want to have a class that keeps some items in a collection, and that allows to enumerate the elements in that collection.

```
Collector
items
itemsDo:
itemsCollect:
itemsSelect:
itemsIsEmpty
...
```

```
itemsDo: aBlock
^items do: aBlock

itemsCollect: aBlock
^items collect: aBlock

... 
```
“Let’s generate these methods statically”

| enumerationSelectors code codeTemplate |
codeTemplate := ‘<1s><n><t>”Generated Automatically”<n><n>
<t>^items <1s>’.

enumerationSelectors := Collection organization
                      listAtCategoryNamed: #enumerating.

enumerationSelectors do: [:selector |
  code := WriteStream on: String new.
  selector keywords with: (1 to: selector numArgs)
  do:[:keyword :nr |
    code nextPutAll: keyword; space;
    nextPutAll: arg; print: nr; space].
  Collector
  compile: (codeTemplate expandMacrosWith: code contents)
  classified: #enumerating]
Let’s forward them to items

Collector>>doesNotUnderstand: aMessage

<table>
<thead>
<tr>
<th>enumerationSelectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumerationSelectors := Collection organization</td>
</tr>
<tr>
<td>listAtCategoryNamed: #enumerating.</td>
</tr>
</tbody>
</table>

^(enumerationSelectors includes: aMessage selector)

ifTrue: [items perform: aMessage selector

          withArguments: aMessage arguments]

ifFalse: [super doesNotUnderstand: aMessage]
Let’s generate on the fly

```smalltalk
doesNotUnderstand: aMessage
  | selector |
  selector := aMessage selector.
  (self isEnumerationSelector: selector)
  ifFalse: [
    ^super doesNotUnderstand: aMessage].
  self compileEnumerationMethodFor: selector.
  ^self perform: selector withArguments: aMessage arguments

isEnumerationSelector: selector
  | enumerationSelectors |
  enumerationSelectors := Collection organization
  listAtCategoryNamed: #enumerating.
  ^enumerationSelectors includes: selector

compileEnumerationMethodFor: selector
  | codeTemplate code |
  codeTemplate := self enumerationTemplate.
  code := WriteStream on: String new.
  selector keywords with: (1 to: selector numArgs)
  do: [:keyword :nr | code nextPutAll: keyword;
    space; nextPutAll: 'arg'; print: nr; space].
  self class
  compile: (codeTemplate expandMacrosWith: code contents)
  classified: #enumerating
```