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

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> Methods, variables and blocks
> Test-Driven Development
> Managing Projects

Selected material courtesy Stéphane Ducasse
Roadmap

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> Methods, variables and blocks
> Test-Driven Development
> Managing Projects
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, Dolt, PrintIt and InspectIt

> **Accept**
  — Compile a method or a class definition

> **Dolt**
  — Evaluate an expression

> **PrintIt**
  — Evaluate an expression and print the result (#printOn:)

> **InspectIt**
  — Evaluate an expression and inspect the result (#inspect)
> 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”

> 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.
Roadmap

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> Methods, variables and blocks
> Test-Driven Development
> Managing Projects
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

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>^</code></td>
<td>return</td>
</tr>
<tr>
<td>&quot;...&quot;</td>
<td>comment</td>
</tr>
<tr>
<td><code>#</code></td>
<td>symbol or array</td>
</tr>
<tr>
<td><code>'...'</code></td>
<td>string</td>
</tr>
<tr>
<td><code>[ ]</code></td>
<td>block or byte array (VisualWorks)</td>
</tr>
<tr>
<td><code>.</code></td>
<td>statement separator</td>
</tr>
<tr>
<td><code>;</code></td>
<td>message cascade</td>
</tr>
<tr>
<td>`</td>
<td>...</td>
</tr>
<tr>
<td><code>:=</code></td>
<td>assignment (also _ or ←)</td>
</tr>
<tr>
<td><code>$_</code></td>
<td>character</td>
</tr>
<tr>
<td><code>:</code></td>
<td>end of selector name</td>
</tr>
<tr>
<td><code>_e_ _r_</code></td>
<td>number exponent or radix</td>
</tr>
<tr>
<td><code>!</code></td>
<td>file element separator (used in change sets)</td>
</tr>
<tr>
<td><code>&lt;primitive: ...&gt;</code></td>
<td>for VM primitive calls</td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td>&quot;a comment&quot;</td>
</tr>
<tr>
<td>Character</td>
<td>$c \ $h \ $a \ $r \ $a \ $c \ $t \ $e \ $r \ $s \ $# \ $@</td>
</tr>
<tr>
<td>String</td>
<td>'a nice string'</td>
</tr>
<tr>
<td>Symbol</td>
<td>#mac #+</td>
</tr>
<tr>
<td>Array</td>
<td>#(1 2 3 (1 3) $a 4)</td>
</tr>
<tr>
<td>Dynamic Array (Pharo)</td>
<td>{ 1 + 2 . 3 / 4 }</td>
</tr>
<tr>
<td>Integer</td>
<td>1, 2r101</td>
</tr>
<tr>
<td>Real</td>
<td>1.5, 6.03e-34,4, 2.4e7</td>
</tr>
<tr>
<td>Fraction</td>
<td>1/33</td>
</tr>
<tr>
<td>Boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>Pseudo Variables</td>
<td>self, super</td>
</tr>
<tr>
<td>Point</td>
<td>10@120</td>
</tr>
</tbody>
</table>

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: ...
Roadmap

> Everything is an Object
> Syntax on a Postcard
> **Three Kinds of Messages**
> Methods, variables and blocks
> Test-Driven Development
> Managing Projects
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*:
  ```smalltalk
  10 bitShift: 2
  ```

- `ifTrue:` (if-then-else) is just *a message sent to a boolean*:
  ```smalltalk
  (x>1) ifTrue: [ Transcript show: 'bigger' ]
  ```

- `do:, to:do:` (loops) are just *messages to collections or numbers*:
  ```smalltalk
  #(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 | 9 (!)
1 + (2 * 3) | 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:

<table>
<thead>
<tr>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 * 3 - 5</td>
</tr>
<tr>
<td>5 &gt;= 7</td>
</tr>
<tr>
<td>6 = 7</td>
</tr>
<tr>
<td>'hello', ' ', 'world'</td>
</tr>
<tr>
<td>(3@4) + (1@2)</td>
</tr>
<tr>
<td>2&lt;&lt;5</td>
</tr>
<tr>
<td>64&gt;&gt;5</td>
</tr>
</tbody>
</table>
Roadmap

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> **Methods, variables and blocks**
> Test-Driven Development
> Managing Projects
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

```Smalltalk
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

\[ ^1 < 2 \text{ ifTrue: } ['smaller'] \text{ ifFalse: } ['bigger'] \]

'smaller'
Variables

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

> **Block parameters** are delimited by `: var |`

```smalltalk
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
```

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

© Oscar Nierstrasz
Control Structures

> Every control structure is realized by message sends

```smalltalk
n := 10.
[n>0] whileTrue:
    [ Transcript show: n; cr.
      n := n-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 (!)

```smalltalk
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*
>     ```smalltalk
>     Node>>accept: aPacket
>     ```
>   — *Class Method* defined in the class *Node class*
>     (i.e., in the class of the the class *Node*)
>     ```smalltalk
>     Node class>>withName: aSymbol
>     ```
> > aSomething is an instance of the class *Something*
Roadmap

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> Methods, variables and blocks
> **Test-Driven Development**
> Managing Projects
Change sets

Make sure your changes are logged to a new change set
A TestSuite bundles a set of Tests

A TestCase contains a number of test* methods

<table>
<thead>
<tr>
<th>TestCase</th>
</tr>
</thead>
<tbody>
<tr>
<td>setUp</td>
</tr>
<tr>
<td>tearDown</td>
</tr>
<tr>
<td>assert:</td>
</tr>
<tr>
<td>deny:</td>
</tr>
<tr>
<td>should:raise:</td>
</tr>
<tr>
<td>shouldn't:raise:</td>
</tr>
<tr>
<td>selector:</td>
</tr>
<tr>
<td>run</td>
</tr>
<tr>
<td>resources</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TestResult</th>
</tr>
</thead>
<tbody>
<tr>
<td>passedCount</td>
</tr>
<tr>
<td>failuresCount</td>
</tr>
<tr>
<td>errorCount</td>
</tr>
<tr>
<td>runCount</td>
</tr>
<tr>
<td>tests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TestResource</th>
</tr>
</thead>
<tbody>
<tr>
<td>isAvailable</td>
</tr>
<tr>
<td>isUnavailable</td>
</tr>
<tr>
<td>setUp</td>
</tr>
<tr>
<td>tearDown</td>
</tr>
</tbody>
</table>

All errors and failures are collected into a TestResult.

Share and reuse (expensive) objects amongst multiple tests.
We will implement the *Money* example in Smalltalk
— First, we develop a test case for a single currency

```
TestCase subclass: #MoneyTest
    instanceVariableNames: 'chf2 chf8 chf10'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Money'
```

*NB:* *This is just a message sent to the TestCase class object (!)*
We will need setters for the private Money state.

```smalltalk
MoneyTest>>setUp
    chf2 := Money new currency: 'CHF'; amount: 2.
    chf8 := Money new currency: 'CHF'; amount: 8.
```
Protocols

Classify methods into *protocols* that reveal their intent
Some obvious tests

```smalltalk
MoneyTest>>testEquals
    self assert: chf2 = chf2.
    self assert: chf2 = (Money new currency: 'CHF'; amount: 2).
    self assert: chf2 != chf8.
```
Money

We define Money as a subclass of Object, with getters and setters

```
Object subclass: #Money
  instanceVariableNames: 'currency amount'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Money'

Money>>currency: aString
  currency := aString.

Money>>currency
  ^ currency

Money>>amount: aNumber
  amount := aNumber.

Money>>amount
  ^ amount
```
Failing tests
Comparisons

Money>>= aMoney
  ^ self currency = aMoney currency
  and: [ self amount = aMoney amount ]

Money>>!= aMoney
  ^ (self = aMoney) not
Constructors

MoneyTest>>testEquals

    self assert: chf2 = chf2.
    self assert: chf2 =
        (Money currency: 'CHF' amount: 2).
    self assert: chf2 != chf8.

We need a constructor on the class side of Money
Class methods

**NB:** Which "self" is referred to in the method body?
Addition

\begin{verbatim}
MoneyTest>>testAdd
  self assert: chf2 + chf8 = chf10
\end{verbatim}

\begin{verbatim}
Money>>+ aMoney
  ^ Money currency: self currency
    amount: self amount + aMoney amount
\end{verbatim}

And so on …
> You can “file out” all your changes so they can be loaded into another image
Change Sets

'Money

instanceVariableNames: 'currency amount'
classVariableNames: ''
poolDictionaries: ''
category: 'Money'!

TestMoney

instanceVariableNames: 'chf2 chf8 chf10'
classVariableNames: ''
poolDictionaries: ''
category: 'Money'!

Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:21'!

!= aMoney
^ (self = aMoney) not! !

Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:26'!
+ aMoney
  self assert: [ self currency = aMoney currency ].
  ^ Money currency: self currency amount: self amount + aMoney amount! !

Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:20'!
= aMoney
  ^ self amount = aMoney amount and: [ self currency = aMoney currency ]! !

...
Creating new instances of `Money` is still cumbersome — We would like to be able to write instead code like this:

```smalltalk
TestMoney>>setUp
    chf2 := 2 chf.
    chf8 := 8 chf.
    chf10 := 10 chf.
```
Extension protocols

To extend an existing class for a category `MyProject`, define the protocol `*MyProject` (or `*myproject`) for that class.
Roadmap

> Everything is an Object
> Syntax on a Postcard
> Three Kinds of Messages
> Methods, variables and blocks
> Test-Driven Development
> Managing Projects
Squeak Examples

Project Description
Examples for the Smalltalk course http://www.iam.unibe.ch/~scg/Teaching/Smalltalk/index.html

Members
Creator: Oscar Nierstrasz
Admin: Oscar Nierstrasz

Registration
MCHttpRepository
    location: 'http://www.squeaksource.com/SqueakExamples'
    user: ''
    password: ''

Links
http://www.squeaksource.com/SqueakExamples.html
http://www.squeaksource.com/SqueakExamples

Statistics
Registered: 19 March 2006 3:59:41 pm
Total Releases: 0
Total Versions: 3
Total Downloads: 5

Version 1.3

© Oscar Nierstrasz
Categories, Projects and Packages

> A system **category** MyProject (and possibly MyProject-*) contains the classes of your application

> A Monticello **package** MyProject contains the categories MyProject and MyProject- *
   — **NB:** If you have sub-categories, keep the top one empty!

> A SqueakSource **project** MyProject stores everything in the Monticello package MyProject
Loading a project

> To load a project from SqueakSource.com
  — Find the project in SqueakSource
  — Open the Monticello Browser
  — Add an HTTP Repository
    - Accept the code fragment from the SqueakSource page
  — Open the repository and load the latest version
Loading a project

HTTP Repository:
location: 'http://www.squeaksource.com/SqueakExamples'
user: 'on'
password: '*****'

Repository: http://www.squeaksource.com/SqueakExamples

Name: SqueakExamples-on.20
Author: on
Time: 28 October 2007, 3:51:35 pm
UUID: 58d2c458-e2a9-4758-b33b-03d978195a99
Ancestors: SqueakExamples-on.19

Added comments to metaclass hierarchy test
Creating a project

To create a project on SqueakSource.com

— Define your categories MyProject or MyProject-*
— Create a SqueakSource project named MyProject
— Define a Monticello package MyProject
— Add an HTTP Repository for MyProject
   
   — Accept the code fragment from the SqueakSource page
— Save the package
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?
Can you answer these questions?

- 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?
- How can you discover the class of a GUI object?
- How does SUnit differ from JUnit?
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.