3. Standard Classes
Reify everything — by reifying its entire implementation model, Smalltalk succeeds in being open, and extensible. New features can be added without changing the syntax!
Roadmap

> Object
> Numbers, Characters, Strings and Arrays
> Variables
> Blocks and Control structures
> Collections

Selected material courtesy Stéphane Ducasse
Roadmap

> **Object**
> **Numbers, Characters, Strings and Arrays**
> **Variables**
> **Blocks and Control structures**
> **Collections**
Review — Objects in Smalltalk

> **Everything** is an object
  — Things only happen by message passing
  — Variables are dynamically bound

> Each object is an instance of one class
  — A class defines the structure and the behavior of its instances.
  — Single inheritance
  — A class is an instance of a metaclass

> Methods are public
  — private methods by convention in “private” protocol

> Objects have private state
  — Encapsulation boundary is the object
Object

> **Object** is the root of the inheritance tree (well, almost)
>  — Defines the common and minimal behavior for all the objects in the system.
>  — Comparison of objects:
>    - `==`, `~~`, `=`, `=~`, `isNil`, `notNil`
>  — Printing
>    - `printString`, `printOn: aStream`
Identity vs. Equality

>  == tests Object identity
    — Should never be overridden

>  = tests Object value
    — Should normally be overridden
      - Default implementation is == !
    — You should override hash too!

'foo','bar' = 'foobaa'
'foo','bar' == 'foobaa'  
true
false
Printing

> Override `printOn:` to give your objects a sensible textual representation

```plaintext
Fraction>>printOn: aStream
    aStream nextPut: $(.
    numerator printOn: aStream.
    aStream nextPut: $/.
    denominator printOn: aStream.
    aStream nextPut: $).
```
# Object methods to support the programmer

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>error: aString</code></td>
<td>Signal an error</td>
</tr>
<tr>
<td><code>doesNotUnderstand: aMessage</code></td>
<td>Handle unimplemented message</td>
</tr>
<tr>
<td><code>halt, halt: aString, haltIf: condition</code></td>
<td>Invoke the debugger</td>
</tr>
<tr>
<td><code>subclassResponsibility</code></td>
<td>The sending method is abstract</td>
</tr>
<tr>
<td><code>shouldNotImplement</code></td>
<td>Disable an inherited method</td>
</tr>
<tr>
<td><code>deprecated: anExplanationString</code></td>
<td>Warn that the sending method is deprecated.</td>
</tr>
</tbody>
</table>
Roadmap

- Object
- Numbers, Characters, Strings and Arrays
- Variables
- Blocks and Control structures
- Collections
Numbers

- Object
  - Magnitude
    - Number
      - Float
      - Fraction
      - Integer
        - SmallInteger
        - LargePositiveInteger
        - LargeNegativeInteger
Abstract methods in Smalltalk

```smalltalk
Number>>+ aNumber
    "Answer the sum of the receiver and aNumber."

    self subclassResponsibility
```
Abstract methods (part 2)

Object>>subclassResponsibility

"This message sets up a framework for the behavior of the class' subclasses. Announce that the subclass should have implemented this message."

self error: 'My subclass should have overridden ', thisContext sender selector printString
Automatic coercion

1 + 2.3
1 class
1 class maxVal class
(1 class maxVal + 1) class
(1/3) + (2/3)
1000 factorial / 999 factorial
2/3 + 1

3.3
SmallInteger
SmallInteger
LargePositiveInteger
1
1000
(5/3)

Browse the hierarchy to see how coercion works.
Try this in Java!

1000 factorial

```
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404
800479988610197196058631666872994808558901323829669944590974245040508073575991882632772188732519779
5059909592761208749754624970436014182780946464962910563938874378864873371191811045825783647849977012
4766328893595573543251318532359846307555740911426241747434934755342864657661166779739666882091207
37914385371598824980812686783874559731374613608537953452422158659320192890878297308431392844430281
2315586110369768013573042161687460976578173483120254785893207616719132448426236131412508780208000261
683151027341827977074814635868170164365024153691398281264810213092761244896359928705114964957419909
34222156682352708082133181611611553615836546984046708975602909050537616475847728421889679646244945
160765353408198930138544248768894595933191017233555602139450399736280750137837615307127761926849034
352625001585835147337161170210396817592151090778801939371871141945452572238655414610628928197860223
838971476085062768629671466746975692291123408243920816015378808899364518562324617616762179168909779
9119037540312746422289988005195444141282012187361745992642956581746628302955570299024324153181617210
46583203678690611712601587835207515156284225504265170483304224216934286933061690899768482590125458327
1682264580665266769958562652272827007575139185817888965208164348348825993266043367660176996162831860
78836150127946559513115655203609398818061213855680301435694527224206344631797460594682573103790084
0244324384465657245201440228188525247093519062092902313649327349756513598720559654228749774011413346
962715524285486562377385230685868897646192738381490014076310146646025989949022221765904339901886
018565264850611697902356193897017860040811889729918311021171298459016141921068884387121855646124960
798722908519296118193288646367582829112132502418664935143970137428531926649875337218904694281
434118520158014123344828010515399429015348307764660937135243272882698646027898864321139083506
21709500259738986355427719674222284785756752344220207573630565949882508796892816275384886339690
959826280956124509948717012445164612637092930912089068942028510640182154399457156805941872748998
094254742173582401063677404595745178516082923013535808184009699637252423056085590370062427124341690
004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000000000
Characters

> Characters:

$ a \ $B \ $$ \ _ \ $1

> Unprintable characters:

Character space, Character tab, Character cr
Strings

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

'mac'
'12'
'A'
'$'
'hello world'

> To introduce a single quote inside a string, just double it.
```
Comments and Tips

> A comment can span several lines.
  — Avoid putting a space between the " and the first character.
  — When there is no space, the system helps you to select a commented expression. You just go after the " character and double click on it: the entire commented expression is selected. After that you can printIt or dolt, etc.

"TestRunner open"
Literal Arrays

#('hello' #(1 2 3))

#(a b c)

#('hello' #(1 2 3))

#(#a #b #c)
Arrays and Literal Arrays

> Literal Arrays and Arrays only differ in creation time
  — Literal arrays are known at compile time, Arrays at run-time.

> A literal array with two symbols (not an instance of Set)
  
  #("Set new")  

> An array with one element, an instance of Set

  Array with: (Set new)  

  an Array(a Set())
Arrays with {} in Pharo

> \{ \ldots \} \text{ is a shortcut for } \text{Array new} \ \ldots \n
\begin{align*}
\#(1 + 2 \ . \ 3) \\
\{ 1 + 2 \ . \ 3 \}
\end{align*}

Array with: 1+2 with: 3

\begin{align*}
\#(1 \ #+ \ 2 \ \#. \ 3) \\
\#(3 \ 3)
\end{align*}
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.

```ruby
'calvin' = 'calvin'
'calvin' == 'calvin'
'cal','vin' = 'calvin'
'cal','vin' == 'calvin'

#calvin = #calvin
#calvin == #calvin
#cal,#vin = #calvin
#cal,#vin == #calvin
#cal,#vin
(#cal,#vin) asSymbol == #calvin
```

**NB:** Comparing strings is slower than comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.
Roadmap

> Object
> Numbers, Characters, Strings and Arrays
> Variables
> Blocks and Control structures
> Collections
Variables

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

> Assignment binds a name to an object reference
  — Not done by message passing!
  — Method arguments cannot be assigned to!
    - *Use a temporary instead*
  — Different names can point to the same object!
    - *Watch out for unintended side effects*

<table>
<thead>
<tr>
<th>p1  p2</th>
</tr>
</thead>
</table>
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>
Instance Variables

> Private to an object
  — Visible to methods of the defining class and subclasses
  — Has the same lifetime as the object
  — Define accessors (getters and setters) to facilitate initialization
    – *Put accessors in a private category!*
Six Pseudo-Variables

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

<table>
<thead>
<tr>
<th>Pseudo-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>
<tr>
<td>self</td>
<td>Reference to this object Method lookup starts from object’s class</td>
</tr>
<tr>
<td>super</td>
<td>Reference to this object (!) Method lookup starts from the superclass</td>
</tr>
<tr>
<td>thisContext</td>
<td>Reification of execution context</td>
</tr>
</tbody>
</table>
Roadmap

- Object
- Numbers, Characters, Strings and Arrays
- Variables
- Blocks and Control structures
- Collections
All control constructs in Smalltalk are implemented by message passing

- No keywords
- Open, extensible
- Built up from Booleans and Blocks
Blocks

> A Block is a *closure*
  > A function that captures variable names in its lexical context
  >  > *i.e., a lambda abstraction*
  >  > First-class value
  >  >  > *Can be stored, passed, evaluated*

> Use to delay evaluation

> Syntax:

```
[ :arg1 :arg2 | |temp1 temp2| expression. expression ]
```

> Returns last expression of the block
Block Example

```
|sqr|
sqr := [:n | n*n ].
sqr value: 5
```

25
Block evaluation messages

```
[2 + 3 + 4 + 5] value
[:x | x + 3 + 4 + 5] value: 2
[:x :y | x + y + 4 + 5] value: 2 value: 3
[:x :y :z | x + y + z + 5] value: 2 value: 3 value: 4
[:x :y :z :w | x + y + z + w] value: 2 value: 3 value: 4 value: 5
```
Booleans

```
Object

Boolean

ifTrue:ifFalse:
not
&

True

ifTrue:ifFalse:
not
&

False

ifTrue:ifFalse:
not
&
```
How would you implement \texttt{not}, \& \ldots?
true and false

> true and false are unique instances of True and False
  — Optimized and inlined

> Lazy evaluation with and: and or:

```
false and: [1/0]  false
false & (1/0)  ZeroDivide error!
```
Various kinds of Loops

|n|
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 ]

10 timesRepeat: [ Transcript show: 'hi'; cr ]

In each case, what is the target object?
Exceptions

-1 factorial

Error!

[:n |
  [n factorial]
  on: Error
  do: [0]
] value: -1

0
Roadmap

- Object
- Numbers, Characters, Strings and Arrays
- Variables
- Blocks and Control structures
- Collections
Collections
The Collection hierarchy offers many of the most useful classes in the Smalltalk system
— Resist the temptation to program your own collections!

Classification criteria:
— Access: indexed, sequential or key-based.
— Size: fixed or dynamic.
— Element type: fixed or arbitrary type.
— Order: defined, defineable or none.
— Duplicates: possible or not
# Kinds of Collections

<table>
<thead>
<tr>
<th>Sequenceable</th>
<th>ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayedCollection</td>
<td>fixed size + index = integer</td>
</tr>
<tr>
<td>Array</td>
<td>any kind of element</td>
</tr>
<tr>
<td>String</td>
<td>elements = character</td>
</tr>
<tr>
<td>IntegerArray</td>
<td>elements = integers</td>
</tr>
<tr>
<td>Interval</td>
<td>arithmetic progression</td>
</tr>
<tr>
<td>LinkedList</td>
<td>dynamic chaining of the element</td>
</tr>
<tr>
<td>OrderedCollection</td>
<td>size dynamic + arrival order</td>
</tr>
<tr>
<td>SortedCollection</td>
<td>explicit order</td>
</tr>
<tr>
<td>Bag</td>
<td>possible duplicate + no order</td>
</tr>
<tr>
<td>Set</td>
<td>no duplicate + no order</td>
</tr>
<tr>
<td>IdentitySet</td>
<td>identification based on identity</td>
</tr>
<tr>
<td>Dictionary</td>
<td>element = associations + key based</td>
</tr>
<tr>
<td>IdentityDictionary</td>
<td>key based on identity</td>
</tr>
</tbody>
</table>
Some Collection Methods

- Are defined, redefined, optimized or forbidden (!) in subclasses

<table>
<thead>
<tr>
<th>Accessing</th>
<th>size, capacity, at: anIndex, at: anIndex put: anElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>isEmpty, includes: anElement, contains: aBlock, occurrencesOf: anElement</td>
</tr>
<tr>
<td>Adding</td>
<td>add: anElement, addAll: aCollection</td>
</tr>
<tr>
<td>Removing</td>
<td>remove: anElement, remove: anElement ifAbsent: aBlock, removeAll: aCollection</td>
</tr>
<tr>
<td>Enumerating</td>
<td>do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect: aBlock, detect: aBlock ifNone: aNoneBlock, inject: aValue into: aBinaryBlock</td>
</tr>
<tr>
<td>Converting</td>
<td>asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: aBlock</td>
</tr>
<tr>
<td>Creation</td>
<td>with: anElement, with:with:, with:with:with:, with:with:with:with:, withAll: aCollection</td>
</tr>
</tbody>
</table>
Array example

```st
|life|
life := #(calvin hates suzie).
life at: 2 put: #loves.
life

#(#calvin #loves #suzie)
```

<table>
<thead>
<tr>
<th>Accessing</th>
<th>first, last, atAllPut: anElement, atAll: anIndexCollection put: anElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching</td>
<td>indexOf: anElement, indexOf: anElement, ifAbsent: aBlock</td>
</tr>
<tr>
<td>Changing</td>
<td>replaceAll: anElement with: anotherElement</td>
</tr>
<tr>
<td>Copying</td>
<td>copyFrom: first to: last, copyWith: anElement, copyWithout: anElement</td>
</tr>
</tbody>
</table>
Dictionary example

```plaintext
|dict|
dict := Dictionary new.
dict at: 'foo' put: 3.
dict at: 'bar' ifAbsent: [4].
dict at: 'bar' put: 5.
dict removeKey: 'foo'.
dict keys
```

---

<table>
<thead>
<tr>
<th>Accessing</th>
<th>at: aKey, at: aKey ifAbsent: aBlock, at: aKey ifAbsentPut: aBlock, at: aKey put: aValue, keys, values, associations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Removing</td>
<td>removeKey: aKey, removeKey: aKey ifAbsent: aBlock</td>
</tr>
<tr>
<td>Testing</td>
<td>includeKey: aKey</td>
</tr>
<tr>
<td>Enumerating</td>
<td>keysAndValuesDo: aBlock, associationsDo: aBlock, keysDo: aBlock</td>
</tr>
</tbody>
</table>
Common messages

#(1 2 3 4) includes: 5
#(1 2 3 4) size
#(1 2 3 4) isEmpty
#(1 2 3 4) contains: [:some | some < 0 ]
#(1 2 3 4) do:
    [:each | Transcript show: each ]
#(1 2 3 4) with: #(5 6 7 8)
    do: [:x : y | Transcript show: x+y; cr]
#(1 2 3 4) select: [:each | each odd ]
#(1 2 3 4) reject: [:each | each odd ]
#(1 2 3 4) detect: [:each | each odd ]
#(1 2 3 4) collect: [:each | each even ]
#(1 2 3 4) inject: 0
    into: [:sum :each | sum + each]
Converting

> Send `asSet`, `asBag`, `asSortedCollection` etc. to convert between kinds of collections

> Send keys, values to extract collections from dictionaries

> Use various factory methods to build new kinds of collections from old

```
Dictionary newFrom: {1->#a. 2->#b. 3->#c}
```
Iteration — the hard road and the easy road

How to get absolute values of a collection of integers?

```smalltalk
|aCol result|
aCol := #(-2 -3 4 -35 4 -11).
result := aCol species new: aCol size.
1 to: aCol size do:
    [ :each | result at: each put: (aCol at: each) abs].
result
```

```
#(2 3 4 35 4 11)
```

```
#((-2 -3 4 -35 4 -11)) collect: [:each | each abs]
```

```
#(2 3 4 35 4 11)
```

**NB:** The second solution also works for indexable collections and sets.
Functional programming style

```plaintext
|factorial|
factorial :=
    [:n |
        (1 to: n)
        inject: 1 into:
            [:product :each | product * each ]].

factorial value: 10

3628800
```
What you should know!

- How are abstract classes defined in Smalltalk?
- What’s the difference between a String and a Symbol?
- Where are class names stored?
- What is the difference between self and super?
- Why do we need Blocks?
- How is a Block like a lambda?
- How would you implement Boolean>>and:? 
- What does inject:into: do?
Can you answer these questions?

- How are Numbers represented internally?
- Is it an error to instantiate an abstract class in Smalltalk?
- Why isn’t the assignment operator considered to be a message?
- What happens if you send the message #new to Boolean? To True or False?
- Is nil an object? If so, what is its class?
- Why does ArrayedCollection>>add: send itself the message shouldNotImplement?
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.