This Lesson

- Identifiers
- Variables
- Literal Objects
- Control Structures
- A Small Example
Identifiers

• Variable names
  – Local variable: begin with a lower case letter (convention).
    • `newName := 17` Assignment to local variable
  – Shared variable: begin with an upper case letter (convention).
    • `Transcript` Global variable (defined in smalltalk’s dictionary)

• Class names
  – Begin with a capital letter (convention).
    • `Integer`
Identifiers cont.

- **Pseudo-variables**: reserved names. Behave like local variables except they *cannot be assigned into*.
  - `self`: Current object
  - `super`: Current object as its superclass
  - `thisContext`: The top frame of the run-time stack
  - `true`: Instance of True
  - `false`: Instance of False
  - `nil`: Instance of UndefinedObject

- Also can’t assign into:
  - Method parameters
  - Block parameters
Variables

• Global Variables: live forever.
• Local Variables:
  – Parameters to the block/method
  – Variables defined within the method
  – Example:
    ```
    findSkill: aSkill .
    | empsWithSkill |
    empsWithSkill := Dictionary new.
    ... ^empsWithSkill
    ```
  – Live as long as they’re needed
  – Often obey call semantics
• Instance Variables:
  – Defined in a class definition.
  – One copy per class instance.
  – Live as long as the instance (also a variable) lives.
  – Example:

```ruby
Object subclass: #Animal
  instanceVariableNames: 'isAsleep hungry...'
classVariableNames: ' NumLegs '
poolDictionaries: '
category: 'Zoo'
```
Class Variables

• Class Variables:
  – Shared among all the instances of a class.
  – Live as long as the class lives.
  – Example:
    ```ruby
    Object subclass: #Animal
    instanceVariableNames: ‘isAsleep hungry...’
    classVariableNames: 'NumLegs'
    poolDictionaries: ‘’
    category: ‘Zoo’
    ```

• Pool Variables:
  – Shared between several classes that may not be related by inheritance.
  – Not a good practice – try to avoid them
Self and Super

• **self** indicates the current object itself, for
  – Sending messages belonging to the current object.
  – returning

• **super** indicates the **current object itself**!
  – Used for calling methods defined in the super class.
  – Method lookup starts in the superclass’ methods.
Methods in A:
  foo
  Transcript show: 'A'; cr.

Methods in B:
  foo
  Transcript show: 'B'; cr.

  bar1
  Transcript show: self class; cr.
  self foo.

  bar2
  Transcript show: super class; cr.
  super foo.

b := B new.
b bar1.
b bar2.

Output:
B
B
B
A
Messages - reminder

• Message –
  – A request for an operation sent to an object
  – Can contain argument values

• Method – A code segment in the object
  – What to do when messages are received.
  – Defined in the object’s class
Message Types

• Send a message using the general format:
  \[ \text{receiver selector arguments} \]

• Three types of messages:
  – Unary
  – Binary
  – Keyword
Unary Messages

- Formed by a single word, requires no arguments.
  - $7 \text{ sign}$ Yields 1
  - $7 \text{ factorial}$
  - $7 \text{ sqrt}$

- Evaluated left to right:
  - $4 \text{ factorial sqrt}$ Yields $\sqrt{(4!)^2}$
Binary Messages

• Formed from one or two adjacent nonalphabetic characters,

• A single argument is required.

\[ -7 + 4 \quad \text{Yields 11} \]

\[ -7 + 3 \times 3 \quad \text{Yields 30 – calculated from left to right!} \]

\[ -7 + 4 \, sqrt \quad \text{Yields 9 - Unary message has a higher precedence} \]
Keyword messages

• Consists of one or more keywords
  – Keyword: an identifier followed by a colon
  – Each keyword requires an argument.
  – 7 max: 14  Max between 7 and 14
  – 7 between: 2 and: 4 * 2
    Yields true. Binary message has precedence

• The selector of the method:
  – max:
  – between: and:
Message Precedence

Unary >> Binary >> Keyword

• Example:
  4 squared + 1 negated gcd: 3 factorial
  3

• Use parentheses to change precedence:
  (4 squared + 1 negated gcd: 3) factorial
  6

• Cannot combine keyword messages:
  100 max: 50 max: 102
  ERROR: SmallInteger does not understand message #max:max:
    – Use parenthesis to separate them
  (100 max: 50) max: 102
  102
Cascades

• The character ; may be used to send several messages to the same object.
• The receiver of all cascaded messages is the receiver of the first message involved in a cascade
• The value returned by a cascade of messages is the value of the last cascaded message

Examples:
2+2; * 1000; squared; factorial; sqrt
1.414213562373095
( 'Smalltalk' copyFrom:1 to:5 ) size
5
'Smalltalk' copyFrom:1 to:5; size
9
'Smalltalk' at:4 put:$r; at:5 put:$t; asString
Smarttalk

Why wasn’t the answer Smalltalk?
Literal Objects

A way to quickly declare objects in a unambiguous way.

- **Integer:** 7
- **Float:** 3.14
- **Char:** $A \quad $9 \quad $$
- **String (in Squeak this is a subclass of ArrayedCollection):**
  
  'I am a string'

- **Symbol (a specialized kind of String, used for system names):**
  
  #aSymbol #at:put:

- **Array:**
  
  $\{-1. \ 2. \ 1+2\} \quad \text{array of expressions}$
Numbers

• May be:
  – Integer ( 1, -3 )
  – Float ( 0.25, 3.5e-2 )
  – Fraction ( 1/4, -3/5 )
• Arithmetic messages: + − ∗ /
• Comparison messages: < > = <= >= ~=
• Arithmetic keyword operations: quo: rem:
• Bitwise logical operations: bitAnd: bitInvert:
  bitOr: bitXor: bitShift:
• Other messages: positive negative
  strictlyPositive squared sqrt lcm: gcd:
  abs truncated rounded ceiling floor...
Characters

• Characters are written with a preceding dollar sign: $A$ $9$ $$

• Comparison messages: $<$ $>$ $=$ $<=$ $>=$ $~=$

• `asInteger` returns the numeric character code corresponding to the receiver.

• `asString` returns a string of length 1 corresponding to the receiver.

• `isAlphaNumeric`, `isDigit`, `isLowercase`, `isUppercase`
  – return `true` if the receiver satisfies the condition, `false` otherwise.
Strings

• Sequences of characters enclosed by single apostrophes.
  ‘hello world. I am a string’

• Comparison messages: < > = <= >= ~=

• Concatenation: ,

• Change: at:put:

• Substring: copyFrom:to:
Arrays

Two ways of defining Arrays:

• Literal array (constants):
  – Contains only literal elements.
  – Turns any non-literal into a symbol.
  
  We will not focus on these...

• Generic array (runtime evaluation):
  – Contains any type of element.
  – Elements separated by periods.
  – Evaluates elements when needed.

  \{hello. world\} contains UndefinedObjects because \texttt{hello} and \texttt{world} aren’t recognized

  \{#hello. #world\} contains \#hello and \#world

  \{1+2. ‘a string’\} contains 3 and ‘a string’
Blocks

• Anonymous functions in squeak.
• A block has the general form:
  [:param1 :param2 ... | | args | statements]

• The block:  
  [:x | x + 1]

  can be understood as:  
  \[ f : f(x) = x + 1 \]  or  \[ \lambda x : x + 1 \]

• Return value of a block is the value of it’s last statement:

  [42]  [x := ‘hello’. 1+2]
• Encapsulates a sequence of Smalltalk statements.
  
  ```smalltalk
  [ Transcript show:'hello' ]
  ```

• Several statements are separated by dots.

  ```smalltalk
  [ i := i + 1 . Transcript show: i ]
  ```

• Executes only when received the message `value`.

  ```smalltalk
  [ i := i + 1 . Transcript show: i ]
  ```

• May have parameters.

  ```smalltalk
  [ :x :y | Transcript show:(x + y) ]
  ```

  ```smalltalk
  value: 2  value: 3
  ```

  – Parameters are local.
Blocks as First Class Citizens

• May be stored in variables
• May be returned from methods
• Example
  – twice := [:x | 2 * x]
  – twice value: 10
    • Yields 20
  – twice value: 5
    • Yields 10
Blocks as Closures

• Executes in the context in which it was defined.
  
  ```
  i := 1.
  ...
  b := [ i := i + 1 . Transcript show: i].
  ...
  b value
  ```

  – The identifier `i` above refers to the binding known at the time the block was defined.
  – Even if the context where `i` was defined ends, `i` will exist as long as the block exists

• Closure: a function that may refer to independent (free) variables.
  – ‘remembers’ the environment in which it was created.
Question from Moed A, Winter 2014-15:

Given the method `foo` in `MyClass`:

```
foo
| i  b |
  i := 5.
  b := [i := i * 2. i].
  i := 3.
  ^b
```

What will the following code print?

```
a i b1 b2|
i := 4.
a := MyClass new.
b1 := a foo.
b2 := a foo.
Transcript show: b1 value; cr.
i := 1.
Transcript show: b1 value; cr.
Transcript show: b2 value; cr.
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>
Returning From a Block
(Non-local return)

An explicit return (non-local return) causes a return from the block’s creation point:

```plaintext
setBlock
  self inner: [ ^1 ].

inner: aBlock
  aBlock value.
  "jump to setBlock and return from there"

Outer
|temp|
  temp := self getBlock.
  temp value.
  "error - getBlock already returned"

getBlock
 ^[ ^1 ].
```

OK!

Error!
Control Structures

• Blocks are useful for several control structures:
  – anInteger timesRepeat: aBlock
  – anArray do: aBlock
  – anInterval do: aBlock
  – aBoolean ifTrue: aTrueBlock ifFalse: aFalseBlock
  – aBlock whileTrue: anotherBlock

• Examples:
  – 5 timesRepeat:
    [Transcript show: 'Hello, world!'; cr]
  – #(85 80 75) do: [ :grade | sum := sum + grade ]
  – (0 to: 10 by: 2) do:
    [ :i | Transcript show: i squared; cr]
  – 0 positive ifTrue:
    [Transcript show: '0 is positive'; cr]
  – [ count <= max ] whileTrue:
    [Transcript show: count; cr.
    count := count+1]
In Smalltalk, several objects may be uniquely identified by *literals*. Some examples are:

- **Number**: 1 3.14 -10 0.27e-2
- **Character**: $a$ $&$ $$
- **String**: 'Hello, world!' '9 o''clock'
- **Array**: #('cat' 'dog' 'cow') #( (1 0) (0 1) )
- **Boolean**: true false
- **Block**: [ :name | name print ] [ i := i + 1 ]

Other basic objects may be obtained by sending messages to these literals:

- **Fraction**: Returned by the message `÷` to an *Integer*.
  - Examples: 1/5 -2/3
- **Interval**: Returned by the message `to: by: to a *Number*.
  - Example: 1 to:5 by:2 returns Interval( 1 3 5 )
Defining the Class Point2D

* File point2D.st

Object subclass: #Point2D
    instanceVariableNames: 'xCoord yCoord'
    classVariableNames:"
    poolDictionaries:"
    category: 'Point'!

initialize
    xCoord := 0.
yCoord := 0

!!
    x: newX
    xCoord := newX

!!
    y: newY
    yCoord := newY

!!
    x: newX y: newY
    self x: newX. self y: newY

!!
The Class Point2D (cont.)

\[
x = x^\text{xCoord}
\]
\[
y = y^\text{yCoord}
\]
\[
distanceTo: \ aPoint
\]
\[
= ( ( x^\text{Coord} - \ aPoint^x ) \ squared + \\
( y^\text{Coord} - \ aPoint^y ) \ squared ) \ sqrt
\]
\[
printString
\]
\[
='( ',
   x^\text{Coord} \ printString, ' ', ',
   y^\text{Coord} \ printString,
   ' )'
\]
Using the Class Point2D

(Point2D methodDict keys) do:
    [:x | Transcript show: x asString; show: ' ']
    // distanceTo: initialize y x: printString x:y: y: x
    origin := Point2D new
    ( 0 , 0 )
aPoint := Point2D new.
aPoint x: 3; y: 4
    ( 3 , 4 )
aPoint x + aPoint y
    7
   aPoint distanceTo: origin
    5.0
Summary of Main Points

• Objects
  – Everything is an object.
  – Each object is an instance of a class.
  – Instance variables are private.

• Messages
  – All actions are produced by passing messages.
  – A message activates a method.

• Methods
  – A method has a signature (selector) that defines how it is to be used.
  – $\text{^exp}$ returns the value of exp (an object) as the result.