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

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

• Class names
  – Begin with a capital letter
    • 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
Identifiers Classification

• Reserved Identifiers
  – Self, super, thisContext, true, false, nil

• Predefined Identifiers (None…)

• Library Identifiers
  – Class Names (e.g. Integer, Boolean,…)
  – Global variables (e.g. Transcript)

• Other Identifiers
  – Local variables (e.g. num := 17.)
Variables

• Global Variables: live forever.
• Local Variables:
  – Parameters to the block/method
  – Variables defined within the method
  – Example:

```ruby
findSkill: aSkill . . .
  | empsWithSkill |
empsWithSkill := Dictionary new.
...
^empsWithSkill
```

– Live as long as they’re needed
– Often obey call semantics
public class Counter
{
    private int c;
    private int resetVal;
    public Counter()
    {
        c=0;
        resetVal=0;
    }
    int get() { return c; }
    int getResetVal() { return resetVal; }
    void setDefaultResetVal(int newResetVal){
        resetVal=newResetVal;
    }
    void inc() { ++c; }
    void reset() { c = resetVal; }
}
Object subclass: #Counter
  instanceVariableNames: 'c resetVal'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tutorial1'

reset
  c := resetVal.
resetVal
  ^resetVal.
resetVal: aResetVal
  resetVal := aResetVal.

initialize
  c := 0.
resetVal := 0.

inc
  c := c+1.
Instance Variables

• Instance Variables:
  – Defined in a class definition.
  – One copy per class instance.
  – Live as long as the instance (also a variable) lives.
  – Example:

```plaintext
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 superclass.
  – Method lookup starts in the superclass’ methods.
Self and Super Example

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
Recursion Base of Methods

• Q : We said that all actions are produced by passing messages to objects. Where does the recursion end?

• A : Atomic commands and native methods
  – Assignment
    • b := 5.
  – Return
    • ^5.
  – Native (primitive)
    • 1+2.
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:
  
  \textit{receiver selector arguments}

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

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

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

• Formed from one or two adjacent nonalphabetic characters,
• A single argument is required.

- $7 + 4$ Yields 11
- $7 + 3 \times 3$ Yields 30 – calculated from left to right!
- $-7 + 4 \sqrt{\ }$ 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
Literal Objects

A way to quickly declare objects in a unambiguous way.

- Integer: 7
- Float: 3.14
- Char: $A$ $9$ $$$
- 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} – 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 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’
• Anonymous functions in squeak.

• A block has the general form:
  $$[: \text{param1} : \text{param2} \ldots \mid \mid \text{args} \mid \text{statements}]$$

• The block:
  $$[: \text{x} \mid \text{x + 1}]$$

  can be understood as:
  $$f : f(x) = x + 1 \text{ or } \lambda x : x + 1$$

• Return value of a block is the value of it’s last statement:
  $$[42] \ [x := \text{‘hello’}. \ 1 + 2]$$
Blocks

- 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]
  value
  ```

- May have parameters.

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

6
12
6
Returning From a Block
(Non-local return)

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

```
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
  ^xCoord
!!
y
  ^yCoord
!!

distanceTo: aPoint
  ^ ( ( xCoord - aPoint x ) squared +
      ( yCoord - aPoint y ) squared ) sqrt
!!

printString
  ^'( ',
    xCoord printString, ' ', ',',
    yCoord 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 \(\text{exp}\) (an object) as the result.