# Smalltalk

The best way to predict the future is to invent it.

Alan Kay, 1971

## Topics

- History and significance of Smalltalk
- Object-oriented programming
- The Smalltalk language
- Smalltalk today
- Additional Examples

## History of Smalltalk

- Smalltalk was developed at the Xerox Palo Alto Research Center (PARC) in the early 1970's
- Alan Kay, Dan Engalls, Adele Goldberg, Dave Robson

- Alan Kay coined the term "object oriented"

"…and I can tell you I did not have C++ in mind."

- 2004 Turing Award Winner
- Delivered the Turing Lecture at OOPSLA 2004

## OOP and GUI

- Smalltalk was the first object-oriented programming (OOP) language and the first graphical user interface (GUI)
- Gave Steve Jobs the idea for the MacIntosh

## Steve Jobs, PARC, Dec 1979

And they showed me really three things. But I was so blinded by the first one I didn't even really see the other two. One of the things they showed me was object oriented programming; they showed me that but I didn't even see that. The other one they showed me was a networked computer system...they had over a hundred Alto computers all networked using email etc., etc., I didn't even see that. I was so blinded by the first thing they showed me which was the graphical user interface. I thought it was the best thing I'd ever seen in my life. Now remember it was very flawed, what we saw was incomplete, they'd done a bunch of things wrong. **But we didn't know that at the time but still though they had the germ of the idea was there and they'd done it very well and within you know ten minutes it was obvious to me that all computers would work like this some day.**

## Object-Oriented Programming

- Object: encapsulates data describing something (often an object in the real world) as well as methods (or programs) that manipulate that data
- Message: communication between objects; method invocations
- Class: defines structure and common behavior of a group of objects

## OOP

- Inheritance: reuse of classes by specializing general behavior
- Polymorphism: many different objects can respond to the same message
- Dynamic Binding: resolution of message binding is deferred until runtime

## OOP

- Combination of inheritance with polymorphism and dynamic binding provides a powerful form of genericity
- Allows high levels of reuse, fast development
- BUT: "Even if you only want the banana, you have to take the whole gorilla."

## Smalltalk Implementations

- Smalltalk is written mostly in Smalltalk
- The core of any Smalltalk implementation is a small bytecode interpreter (virtual machine)
- Smalltalk source code is compiled to intermediate bytecode to be executed on the VM

## Programming in Smalltalk

- Programming in Smalltalk involves
  - Defining classes
  - Implementing class methods
  - Evaluating expressions
- Most Smalltalk languages provide an interactive GUI as a development environment

### Objects

- have local memory, inherent processing capability, capability to communicate with other objects.
- Examples:

| | |
|---|---|
| 2 | True |
| FileStream | Text Editor |
| Class | MetaClass |

- Can be passed as parameters and returned as results

## Messages and Methods

- Message: a request to an object to perform an operation (analogous to function call)
- Method: implementation of an operation (function)
- Examples of messages:

```
'hello, world' size
#(1 12 24 36) includes: 4 factorial
3 < 4 ifTrue: ['Yes'] ifFalse: ['No']
```

## Classes and Instances

- Class: The abstract definition of the behavior of a class of objects
- Instance: an individual object defined by a class

## Instance methods and Class methods

- Instance method: a method that describes how an operation is carried out by every instance of a class
- Class method: a method that describes how an operation is carried out by a class, such as creating a new instance of itself.

  Class methods are sent to the class (an instance of Class MetaClass) rather than to an instance of the class

## Instance variables

- Instance variables: a variable that stores data for an instance of a class
- Generally have different values for each instance
- The collection of instance variables describes the state of the object

## Class variables

- A variable that is shared by a class and all of its instances
- Available to all instances of a class
- Can be constants (Float pi) or even references to existing instances, such as Students class that maintains a list of Student instances

## Inheritance

- Classes are arranged in a hierarchy
  - Superclass: the parent of a class
  - Subclass: a child of a class
- Subclasses inherit the variables and methods of the superclass

## Inheritance in Smalltalk

- Is a strict tree structure: a subclass can have one and only one parent class
- Subclasses usually "specialize" the more general behavior of the parent class
  - Can add new variables or methods
  - can also hide inherited functionality

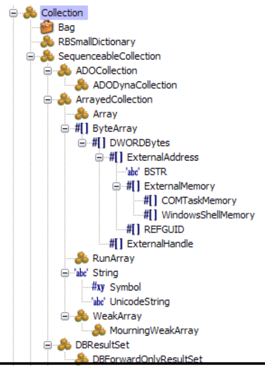
## The Magnitude Class Hierarchy

```
Magnitude
    Association
    Character
    Date
    Number
        Float
        Fraction
        Integer
            LargeInteger
            SmallInteger
    Time
```

## The Collection Class Hierarchy

```
Collection
    Bag
    IndexedCollection
        FixedSizeCollection
            Array
            Bitmap
            ByteArray
                CompiledMethod
        Interval
        String
            Symbol
        OrderedCollection
            Process
            SortedCollection
    Set
        Dictionary
        …
```

## Another Collection Hierarchy



```
Collection
    Bag
    RBSmallDictionary
    SequenceableCollection
        ADOCollection
            ADODynaCollection
        ArrayedCollection
            Array
            ByteArray
                DWORDBytes
                    ExternalAddress
                        BSTR
                        ExternalMemory
                            COMTaskMemory
                            WindowsShellMemory
                        REFGUID
                    ExternalHandle
            RunArray
            String
                Symbol
                UnicodeString
            WeakArray
                MourningWeakArray
        DBResultSet
            DBForwardOnlyResultSet
```

22

## Expressions

- Smalltalk does not have a "statement" in the sense used by procedural languages
- Computation is achieved by evaluating expressions
- All expressions return an object
- Types:
  - Literals
  - Variable names
  - Message expressions
  - Blocks of code

## Literals and Variables

- Literals

```
#aSymbol    #(1 2 4 16 32 64)
'Hello, World'
```

- Variables

```
Smalltalk    x    selectedDictionary
```

## Variables

- Names are syntactically similar to other languages
- All variables are pointers to objects
- All variables are inherently typeless

  No difference between pointing to the number 42 and pointing to an instance of a text editor

## Public and Private Variables

- Public variables are shared and visible globally

  Name begins with upperclass letter
- Private variables are local to an object, block or method
  - Name begins with lowerclass letter

## Message Expressions

- Sending a message involves:
  - object to which message is sent ("receiver")
  - additional objects included in message ("arguments")
  - desired operation to be performed ("message selector")
  - Accepting the single object returned as the "message answer"

## Message Expression Examples

```
set  add: stream next
```

**(receiver is set, selector is add, argument is result of stream next, where receiver is stream and next is selector)**

```
2 + 3
```

**(receiver is 2, selector is +, argument is 3)**

```
array at: index + offset put: Bag new

array at: 1 put: self
```

## Message Syntax Types

- Unary (no parameters)

  ```
  firstAngle cos
  42 PrintString
  ```
- Binary

  ```
  2 + 3
  thisCollection = thatCollection
  ```

## Message Syntax Types

- Keyword (a general extension of infix syntax)

  ```
  myArray at: 42 put: 5
  ```

  (selector is "at: put:")
- In practice very few methods accept more than 2 parameters

## Selector Evaluation Order

- Rules are a bit odd unless you are used to APL
- Unary and binary expressions associate left to right

```
12 - 3 * 3 -> 27
12 - (3 * 3) -> 3
12 - 12 sin * 3 -> 0
```

## Cascading Messages

- A series of messages sent to same object

```
myPen home; up; goto: 100@200; down; home
```

Equivalent to:

```
     myPen home
     myPen up
     myPen goto: 100@200
     myPen down
     myPen home
```

## Method Definitions

- General Syntactic Form

```
MessagePattern
| local variables |
Expressions
```

- Return object designated with ^
- An object can refer to itself with self

## Self references

- Often an object needs to send a message to itself
- Pseudovariable "self" is used to refer to the object itself
- Examples:

```
count = 0
  ifTrue: [self error:
     "0 items being averaged"]
  ifFalse: [ ^sum / count]
```

## Self references

- Fibonacci numbers

```
fibonacci
 "Answer the nth fibonacci number, where
  n is the receiver"
^self < 3
  ifTrue: [1]
  ifFalse: [(self-1) fibonacci +
            (self-2) fibonacci]
```

Testing fibonacci

```
  #(1 2 3 4 5 6 7 10 30)
  collect: [:m | m fibonacci]
```

## Unary method example

```
left
 "return the left subtree of the
 receiver"
 ^ self left
```

## Binary method example

```
= aCollection
  "answer true if elements contained by
  receiver are equal to the elements
  contained by aCollection "
| index |
self == aCollection
  ifTrue: [^true].
(self class == aCollection class)
  ifFalse: [^false].
```

```
index := self size
index ~= aCollection size
  ifTrue: [^false]
[index <= 0]
  whileFalse: [
  (self at: index) = aCollection at: index)
  ifFalse: [^false].
  index := index – 1.]
^true
```

Note: == is equivalence (test for same object) while = is test for equality (equal values)

## Keyword method example

```
setX: xCoord  setY: yCoord setZ: zCoord
"set coordinates for a threeDpoint"
x := xCoord
y := yCoord
z := zCoord
^self
```

## Assignment

- Syntactically similar to other languages, but variables are “typeless”

-
  ```
  | x |
  x := 1
  x := “A String”
  x := Pen new.
  ```

## Control Structures

- Smalltalk has NO conventional control structures
- Control structures are formed by passing block objects as parameters to Boolean objects

## Blocks

- Objects consisting of a sequence of expressions

```
[index := index + 1. sum := sum + index.]
```

- Expression are evaluated when block receives the message “value”

```
[index := index + 1.
 sum := sum + index.] value
```

- Blocks can be assigned to variables and executed by sending the message "value" to the variable.

## Iteration

```
count := 0.
sum := 0.
[ count <= 20 ]
    whileTrue: [ sum := sum + count.
                 count := count + 1 ]
```

- WhileTrue: is a method of Class Boolean
  value is a method of Class Block
  whileTrue: sends message "value" to conditional block
  Conditional block returns object true or false
  If results is true, WhileTrue: evaluates code in parameter block

## Iteration

```
"copy a disk file"
| input output |
input : = File pathName: 'go'.
output := File pathName: 'junk'.
[input atEnd]
  whileFalse: [output nextPut: input next],
input close,
output close
```

44

## For loop (to: by:)

```
"compute the sum of 1/2, 5/8, 3/4, 7/8, 1"
| sum |
sum := 0.
1/2 to: 1 by: 1/8
  do: [ :i | sum := sum + i] .
^sum
```

45

## Selection

- Similar to iteration. Common control structures

```
ifTrue:
if False:
ifTrue: ifFalse:

total = 0
   ifTrue: [average := 0]
   ifFalse: [average := sum / total]
```

- Message **"= 0"** sent to object **total**
  returns either **True** or **False**
  resulting Boolean object is receiver of **ifTrue:** message

## Block Parameters

- Blocks can accept parameters

```
[:x :y | (x * x) + (y * y).]

[:x :y | (x * x) + (y * y).]
  value: 2 value: 3
-> 13
```

## Classes

- All Smalltalk objects are instances of classes (including classes: they are instances of class MetaClass)
- A Class definition has four parts:
  - Class name
  - Superclass name
  - declaration of local (instance) variables
  - a set of methods that define how instances will respond to messages

## Classes and Message Lookup

- A message sent to an object causes lookup in class definition
- If search fails, then lookup proceeds to superclass
- Top of hierarchy is class Object
- If no method found, error results

## Polymorphism

- Polymorphism: a specific message may be sent to different instances of different class at different times
- Method lookup in Smalltalk occurs at execution time
- Allows unlimited polymorphism since any class can implement any method

## Polymorphism

- Consider **`a + b - c`**

  The methods + and - could be implemented by class String with an arbitrary meaning.
- **`a + b`**
  - string concatenation-returns a with b concatenated
- **`b – c`**
  - returns b with all occurrences of c removed
- Then **`a + b - c`** would first remove all occurrences of c from b, then concatenate the result to a (if a, b and c are instances of String)

## Type Checking

- Variables are typeless and can be bound to any object
- Type checking is performed dynamically when message sent to object
- If object lacks a method to respond to the message, lookup proceeds to superclass

  All the way up to class Object

  Failure at this point results in error

## Inheritance

- A subclass inherits all of the instance variables, instance methods and class methods of the superclass
  - New instance variables can be added
  - Names must differ from names of instance variables in ancestor classes
- A subclass can define new methods or redefine methods implemented in an ancestor class
- Redefinition hides definition in ancestor class
- Pseudovariable "super" can be used in methods to refer method search to the superclass

## A Stack

```
class name                    Stack
superclass name               Object
instance variable names stack

"Class Methods"
new
"Create an instance"
stack = OrderedCollection new.
^self

"Instance Methods"
pop
"Answer the receiver with last item removed from
  stack"
self size = 0
  ifTrue: [self error: "Attempt to pop empty
  stack"]
  ifFalse: [stack removeFirst. ^self]
```

```
push: anItem
"Answer the receiver with anItem added to the
  top"
stack addFirst: anItem.
^self

top
"Answer the top item on the receiver"
^self size = 0
  ifTrue: [self error: "Attempt to take top of
  empty stack"]
  ifFalse: [^stack at: 1]

empty
"Answer true if the receiver is empty"
^self size = 0

size
"Answer the size of the receiver"
^stack size
```

## Another Stack

```
class name                    Stack
superclass name               OrderedCollection
instance variable names none
"Class Methods"
new
"Create an instance"
^super new.
pop
^self removeFirst
push: anItem
^self addFirst: anItem
top
^self at: 1
empty
^self size = 0
```

## Designing a class hierarchy is not easy…

- Problem with stack implementation as subclass of OrderedCollection:

```
myStack:= Stack new.
1 to: 10 do: [ :x | myStack push: x]
middle := myStack at: 5
```

- Need to hide “inappropriate” methods of OrderedCollection

## Smalltalk Today

- "The best way to predict the future is to invent it.“ (Alan Kay, 1971)
- Smalltalk is the origin of two of the most important computing “developments” of the last two decades:
  - Object Oriented Programming (OOP)
  - Graphical User Interfaces (GUIs)

## Smalltalk Today

- Smalltalk still has a small but enthusiastic following
- Often used in industry as a rapid prototyping environment
- IBM (VisualAge Smalltalk) promotes it as an e-commerce tool
- OOVM has developed a Smalltalk VM for embedded systems that does not need an OS
- Implemented in MS .NET 2003 as S#

## Smalltalk Resources

- Most prominent open-source version is Squeak Smalltalk
- See http://www.squeak.org/
- Smalltalk.org( http://www.smalltalk.org )
  has a comprehensive list of products and many interesting articles and tutorials

## Of interest to Ruby users:

- Ruby The Smalltalk Way

http://www.sapphiresteel.com/Ruby-The-Smalltalk-Way

61

## 99 Bottles of Beer

```
"Copy into a workspace, highlight the code and choose do
   it."
"Tested under Squeak 3.7 and VisualWorks 7.3"
| verseBlock |
verseBlock := [ :bottles  | | verse |
  verse := WriteStream with: (String new).
    bottles = 0 ifTrue:
      [verse
        nextPutAll: 'No more bottles of beer on the wall.
   No more bottles of beer...'; cr;
        nextPutAll: 'Go to the store and buy some more...
   99 bottles of beer.'; cr].
    bottles = 1 ifTrue:
```

62

```
[verse
      nextPutAll: '1 bottle of beer on the wall. 1 bottle of
  beer...'; cr;
      nextPutAll: 'Take one down and pass it around, no more
  bottles of beer on the wall'; cr].
   bottles > 1 ifTrue:
     [verse
       nextPutAll: bottles printString; nextPutAll: ' bottles
  of beer on the wall. ';
       nextPutAll: bottles printString; nextPutAll: ' bottles
  of beer...'; cr;
       nextPutAll: 'Take one down and pass it around, ';
       nextPutAll: (bottles - 1) printString, ' bottle';
       nextPutAll: (((bottles - 1) > 1)
              ifTrue: ['s '] ifFalse: [' ']);
       nextPutAll: 'of beer on the wall'; cr].
 verse contents].

99 to: 0 by: -1 do:
   [: i | Transcript show: (verseBlock value: i); cr].
```

- For the true OO version of 99 Bottles, see

http://www.99-bottles-of-beer.net/language-smalltalk-1513.html

63

## Example: Polynomials

- Polynomials
  - Represent Polynomials such $3x^2 + 5x - 7$
  - Representation is a collection of coefficients: #(-7 5 3)
  - Subclass of Magnitude

64

## Polynomial class

```
Magnitude subclass: #Polynomial
        instanceVariableNames: 'coefficient'
        classVariableNames:: ''
        poolDictionaries: ''

new
        "Unary class constructor: return 0*x^0"
        ^ self new: #( 0 )

new: array
        "Keyword class constructor"
        ^ (super new) init: array

init: array
      "Private: initialize coefficient"
      coefficient := array deepCopy
```

65

```
degree
      "Highest non-zero power"
      ^ coefficient size - 1

coefficient: power
      "Coefficient of given power"
      (power >= coefficient size) ifTrue: [ ^ 0 ].
      ^ coefficient at: power + 1

asArray
        ^ coefficient deepCopy

= aPoly
        ^ coefficient = aPoly asArray

!= aPoly
        ^ (self = aPoly) not

< aPoly
        "not defined"
        ^ self shouldNotImplement
```

66

## Evaluate method

```
evaluate: aPolynomial x: aNumber
 "Return the results of evaluating aPolynomial
  for the value aNumber"
  | index val |
  index := 1
  val := 0
  [index < coefficient size] whileTrue:
    [ val := val + (coefficient at: index) *
      (aNumber raisedToInteger: (index – 1)) ]
  ^ val
```

67

## Complex class

```
Object Subclass: #Complex
    instanceVariableNames: 'realpart imagpart'
    classVariableNames:: ''
    poolDictionaries: ''
new
       "Unary class constructor: Invalid"
        ^ self error 'use real:imaginary:'
new: aComplex
        "Class constructor aComplex"
        ^ (super new) copy: aComplex

real: r imaginary: i
  "Class Constructor"
  ^ (super new) setReal: r setImaginary: i

setReal: r setImaginary: i
  "Private instance method to initialize self"
  realpart := r
  imagpart := i
  ^self
```

68

```
real
  "Return real part"
  ^ realpart

imaginary
  "Return imaginary part"
  ^ imagpart

+ val
  "Return new complex number: self + val"
  ^ Complex real: realpart + val real
          imaginary: imagpart + val imaginary

- val
  "Return new complex number: self - val"
  ^ Complex real: realpart - val real
          imaginary: imagpart - val imaginary
```

69

```
negated
  "Return new complex number: - self"
  ^ Complex real: realpart negated imaginary:
  imagpart negated

= val
  "Return self = val"
  ^ (realpart = val real) & (imagpart = val
  imaginary)

< val
  "Not mathemtically defined"
  ^ self shouldNotImplement
```

70