Smalltalk and pure object oriented programming

Greg J. Badros
badros@cs.washington.edu
University of Washington, Seattle
CSE-341, Summer 1999
(C) 1999, Greg J. Badros—All Rights Reserved

OO Focuses on the Data

"It is better to have 100 functions operate on 1 data structure 10 functions on 10 data structures"
—Alan Perlis

Pure Object-Orientation: Everything is an object

An object:
- Encapsulates state
- Has behaviour

More about objects

An object is an instance of a class:
- Encapsulates state
  - instance variables (or fields)
  - private data
- Has behaviour
  - methods, or member functions
  - operates on arguments and private state
  - produces an answer (the return value)

What is a class?

A class is a description of a set of objects
- What makes up those objects?
  i.e., what are its instance variables
- What messages do the objects understand?
  i.e., what methods are implemented

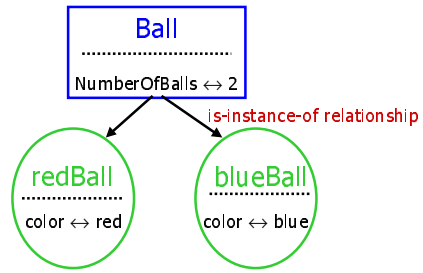
Classes are also objects!

- They have state
  - class variables
  - shared among all instances of the class (and the class object itself!)
- They have behaviour
  - class methods
  - operates only on arguments and class variables

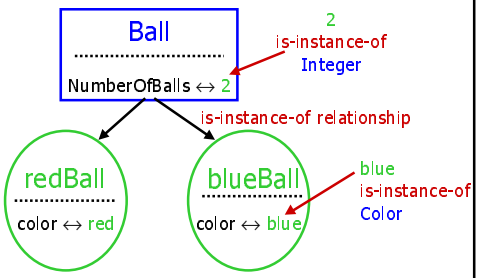
Example

- Class Ball
  class variables: NumberOfBalls
  instance variables: color
- Object redBall is-an-instance-of Ball
- Object blueBall is-an-instance-of Ball

Diagram of Ball class and its instances



Everything is an object!



is-instance-of relationship

OBJECT is-instance-of CLASS
e.g., redBall is-instance-of Ball

- relationship is between an instance object and a class object
- OBJECT is an instance of CLASS, and thus has the instance variables defined by CLASS, and understands messages implemented for CLASS

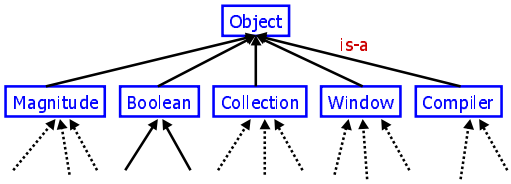
is-a relationship

SUBCLASS is-a SUPERCLASS
e.g., BouncyBall is-a Ball

- relationship is between two classes
- "the subclass relationship"
- instances of BouncyBall can be used like instances of Ball, except... and it also...

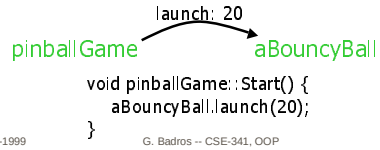
Object hierarchy

Everything is an object!



Computation via messages and methods

- An object sends a message to request an answer from an object (maybe itself!)
- An object that receives a message must implement a method for that message



Object-oriented vs. procedural terminology

Object-Oriented	Procedural
method	procedure
message	procedure invocation
answer	return value
object	N/A
class	N/A

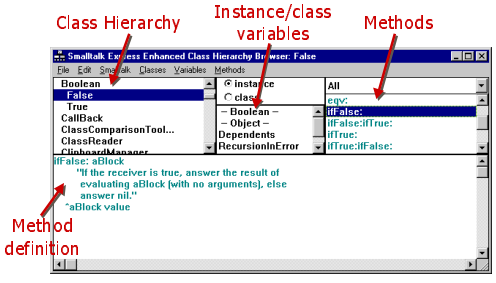
Smalltalk-80

- Preceded by Smalltalk-72, -76
- Based on Simula-67, inspired by Lisp
- Designed by Alan Kay, for children
- Primary Reference: Goldberg and Robson's *Smalltalk-80 The Language*
- Still very popular—especially prevalent in prototyping systems and research

Features of Smalltalk-80

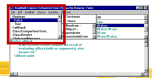
- Incredibly rich programming environment!
  - Not about files—very dynamic system
  - Aids metaphor since objects are more concrete
- Everything is an object
  Single rooted object hierarchy
- Simple syntax
- Closures, lexical scoping, dynamic-typing, automatic garbage collection

Smalltalk Class Browser



Class Hierarchy Pane

The Class Hierarchy Navigator



- Displays a tree showing the names of the classes and their relationships
- Highlighted class has more information displayed in other two panes
- Use **Classes → Find Class...** to locate a class (e.g. *pen*)
- Use **Classes → Browsed Class History** to choose among classes you've visited

Variables Pane

A Filter for the Methods Pane

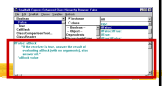
- Radio button selector chooses between instance variables and class variables
- Variables from superclasses are inherited and displayed after, e.g., "- Object -" lines
- Selecting a variable changes the rightmost pane to show only methods that use that variable (**Variables** menu controls "use")
- Click instance/class radio button to show all methods again

Methods Pane

A Listing of the Relevant Methods

- Shows all instance/class methods defined for the class in the leftmost pane, perhaps filtered to list only those using a variable
- Drop-down box can be used to categorize methods into protocols (we'll ignore this)
- Class methods are red
  Instance methods are greenish

Method Definition Window



- Shows the source code for the method highlighted in the Method Pane
  (or information about class if no method is highlighted)
- Use it to read/write/edit method definitions
- Can also use it to add instance/class variables by clicking on a class name in the leftmost pane and then editing the class description that appears in this window

So where's my program?

- Smalltalk "image" is a memory-dump of the state of your Smalltalk system
  - It is the preferred state-saving technique when you're working alone
  - Binary format, implementation dependent
- You must "File out" classes that you want to share—creates a readable text file
  - to turn in code
  - to share with co-developers, et al.

Filing out your code

- **Classes→File Out...** to list the description of only the highlighted class
- Classes→**File Out All...** to list the descriptions of that class and all of its subclasses
- Want all your code?
  Select Object and do Classes→**File Out All...**
  Tip: Do **not** print or turn-in this!

Filing in code

Smalltalk→File It In works on the text that is selected (i.e., highlighted)

- **File→Open** the file you want to file in
- Use **Partial File→Read Entire File** if a big file
- **Edit→Select All** to highlight text (or C-a)
- Finally do **Smalltalk→File It In**
- Then be sure to **Classes→Update** in browsers

Asking questions about code

- Learning class libraries, others' code is an important and under-taught skill
- Often want to be able to find answers to specific questions to help test hypotheses about the code
- Smalltalk environment lets you ask, e.g.:
  - Who implements a method "ifTrue:"?
  - Who sends the message "initialize"?

Learning more about Smalltalk/Express Environment

- Explore the menus
- Double-click on things—experiment!
- Right-click to get context-sensitive menus, and try things!
- Do not save your image after making experimental changes—make changes you want to keep during a different session (but only run one at a time)

Simple Smalltalk Syntax

Smalltalk	Scheme
"comment"	;;;
'string literal'	"..."
$a (char. literal)	#?a
#symbol	'symbol
nil	kind of (), or #f
true false	#t #f
#(2 $a 'foo')	#(2 #?a "foo")

More Smalltalk Syntax

Smalltalk	C++
^ *expression*	return *expression* ;
:=	=
| i j k | (local vars)	int i; double j; char *k;
self	this
myBall stop	myBall.stop();
myBall bounce: 10	myBall.bounce(10);

Stack Object Implementing a Stack ADT

For a stack, we need to be able to:

- Create a new stack
- Pop an element from a stack
- Push an element onto a stack
- Print a stack onto a Stream object

Stack Object Methods

For a stack, we need to be able to:

- Create a new stack ← Class method (a constructor)
- Pop an element from a stack
- Push an element onto a stack
- Print a stack onto a Stream object

} Instance methods

Adding a subclass

- File-out format:

```
Object subclass: #Stack
instanceVariableNames: 'anArray top'
classVariableNames: ' '  poolDictionaries: ' '
!
```

- Interactively:
  - Highlight Object
  - Classes→Add Subclass...

Adding a class method

```
! Stack class methods !
new
    | s |
    s := super new.
    s setSize: 10.
    ^ s.
!
```

Just part of the file-out format

Method definition

Interactively, highlight Stack, ● class

- Right-click in method pane, New method...
- Enter method definition
- Right-click in method definition window, Save

Adding an instance method

```
! Stack methods !
pop
    | item |
    item := anArray at: top.
    top := top – 1.
    ^item
!
```

Interactively, highlight Stack, ● instance

- Right-click in method pane, New method...
- Enter method definition
- Right-click in method definition window, Save

More stack methods

```
push: item
    top := top + 1.
    anArray at: top put: item

setSize: n
    anArray := Array new: n.
    top := 0
```

Period is just a separator, not a terminator

(no period needed here)

Initializing method

Shared Variables

Just by convention

Three kinds, all start with UPPERCASE letter

- Class variables
  Shared by all instances of a single class
  e.g., NumberOfBalls
- Globals (generally avoid these when you can!)
  Shared by all instances of all classes,
  e.g.,Transcript, Turtle
- Pool variables
  Shared by instances of some subset of classes

Private variables

Two kinds

- Instance variables
  Only object's methods and its subclass's methods may read/set them
  "Protected" in C++ terminology
- Temporary variables
  Between | and |, message arguments
  Exist only for method invocation

Variables

- Declared without a type
  - ⇨ Smalltalk, like Scheme, is dynamically typed
- Declared without a binding
  - ⇨ Would not make sense without assignment
- Automatically initialized to **nil**
  - Similar to NULL in C++
  - **nil** understands no messages

Constructing a Stack

C++:

```
Stack s = new Stack();
// Stack::Stack() is a constructor that
// initializes memory that new allocated
```

Smalltalk:

```
s := Stack new.
```

"Stack's new method calls super new to create a new Object, then calls **setSize:** to initialize the instance fields."

Object construction in Smalltalk

- Typical pattern in class method new:

```
| answer |
answer := super new.
answer initialize.
^ answer.
```

Create instance of super class

Was setSize: in Stack example

- Class methods can only send messages to the instances they create—they cannot set/read instance fields

Initializer method

- Should set the state of the object to maintain any invariants that the other methods expect
- Almost always should set all instance variables of the object to a known value
  - Often involves sending the message new to other classes to create instances of them
  - Instance fields will be **nil** if you do not set them

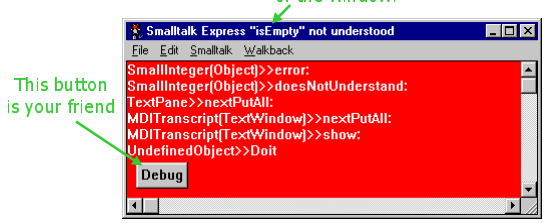
Common mistakes

- Forgetting **^** to return a value—
  self is implicitly returned when a method definition does not explicitly return a value
- Using **=** instead of **:=** for assignment
  Does equality test instead—
  an instance of True or False will get a message not understood error...

Errors...

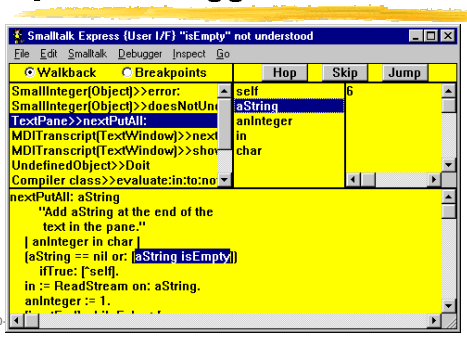
Transcript show: 6.

Read the titlebar of the window!



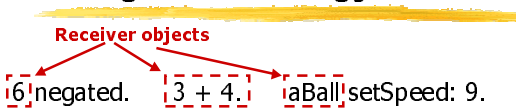
This button is your friend

Superb Debugger!



30- 42

Message terminology



- A ***message*** is sent to a ***receiver*** object to execute a ***method***
- Messages are like function applications or procedure calls
- Methods are like functions or procedures

30-July-1999 G. Badros -- CSE-341, OOP 43

Three kinds of Messages

- Unary—alphanumeric, no arguments.
  6 negated. Stack new.
- Binary—operators (1 or 2 non-alphanumeric)
  3 + 4. 45 @ 50.
- Keyword—1 or more arguments.
  anArray at: 5 put: 'Hello world'.
  aBall setSpeed: 20.

30-July-1999 G. Badros -- CSE-341, OOP 44

Parsing Messages

highest

lowest

Precedence

- Parenthesized expressions
- Unary messages (left associative)
- Binary messages (left associative)
- Keyword messages

2 * theta sin "Calculates 2*sin(theta)."

(2 * theta) sin "Calculates sin(2*theta)."

30-July-1999 G. Badros -- CSE-341, OOP 45

Parsing keyword messages

frame scale: factor max: 5

- 2-argument keyword message with selector **`scale:max:`**

frame scale: (factor max: 5)

- Two 1-argument keyword messages with selectors **`scale:`** and **`max:`**

30-July-1999 G. Badros -- CSE-341, OOP 46

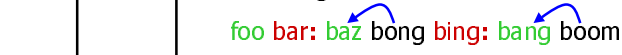
The importance of the colon:

foo bar baz bong bing bang boom
(((((foo bar) baz) bong) bing) bang) boom

- All unary messages

foo bar: baz bong: bing bang boom

- bar:bong:



foo bar: baz bong bing: bang boom

- bar:bing:

30-July-1999 G. Badros -- CSE-341, OOP 47

Cascading messages

```
anArray at: 1 put: 'Hello'.
anArray at: 2 put: 'world'.
```

```
anArray at: 1 put: 'Hello';
        at: 2 put: 'world'.
```

Send another message to the same receiver object

```
6 negated; negated.  ⇒  –6
6 negated negated    ⇒  6
```


Message dispatch

- Choosing the method to run for a given message send is called ***dispatching***
- The most-specialized method is chosen
  - ColoredBall is-a Ball
  - aBall draw. "Runs Ball's draw method"
  - aColoredBall draw. "Runs ColoredBall's draw method"

Dispatch examples

Ball (is-a) ← ColoredBall

Ball: draw, bounce, setSize:

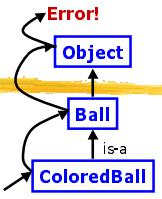
ColoredBall: draw, setColor:

aBall draw.	Ball's
aBall bounce.	Ball's
aBall setSize: 10.	Ball's
aBall setColor: blue.	Message not understood!
aColoredBall draw.	**ColoredBall's**
aColoredBall bounce.	Ball's
aColoredBall setSize: 10.	Ball's
aColoredBall setColor: blue.	ColoredBall's

Method terminology

- ColoredBall's draw method ***overrides*** Ball's draw method
- ColoredBall ***inherits*** its **bounce** and **`setSize:`** methods from Ball

Dispatch algorithm



aColoredBall foo.

Dispatching message **foo**

First look in receiver object's class—
if no method named **foo** is defined there,
look in its superclass, and repeat until an
ancestor class defines that method.
Not found? ⇒ message not understood

Talking to yourself

```
aBall bounce.
anotherBall bounce.
```

Want this to be the object that is receiving the bounce message (here, aBall or anotherBall)

```
"Ball's bounce method"
bounce
  aBall changeDirectionForBounce.
  aBall changeSpeedForBounce.
```

Smalltalk schizophrenia

```
aBall bounce.
anotherBall bounce.

"Ball's bounce method"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.
```

self keyword is like **this** in C++, but **self** is always explicit when sending messages

Writing draw without message dispatch

```
void draw(Object o) {
    if (O.isInstanceOf(Ball)) {
        /* Ball's draw method code */
    } else if (O.isInstanceOf(ColoredBall)) {
        /* ColoredBall's draw method code */
    } else {
        /* error, message not understood */
    }
}
```


So what's the big deal?

- We can write a draw function that works differently on Balls and ColoredBalls
- So who cares about dispatch?

⇨Method dispatch permits keeping the code for each object separated from the code for other objects: modularity!

Modularity

- Consider adding a new class, BouncyBall is-a Ball
- Without OO dispatch, we would have to edit the single monolithic **`draw`** function to make it work on our new class
- Instead we just write our new method, and let the language do the work

⇨Pretend that the language writes a big switch statement for each method name!

Overriding a method

Redefining a method in a subclass is called ***overriding*** that method

Two possibilities

- Replace the method completely
- Augment or extend the old behaviour

Replacing a method

```
"Ball's method"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.

"DeadTennisBall is-a Ball"
bounce
  self setSpeed: 0.
```

Overrides and replaces

A less drastic change in behaviour

"Ball's method"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.

"ColoredBall is-a Ball"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.
  self setColor: blue.

Overrides and replaces

Do not duplicate code!

"Ball's method"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.

"ColoredBall is-a Ball"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.
  self setColor: blue.

Overrides and replaces

identical code

Getting help from your parent

"Ball's method"
bounce
  self changeDirectionForBounce.
  self changeSpeedForBounce.

"ColoredBall is-a Ball"
bounce
  super bounce.
  self setColor: blue.

Overrides and extends

Do the superclass's behaviour

super keyword

- Similar to **self** in that it represents a send of the message to the receiver object
- Difference:
  Using **super** alters the dispatch algorithm to start searching for the method at the superclass of the class that defined the method containing the super keyword (*not* the superclass of the receiver object)

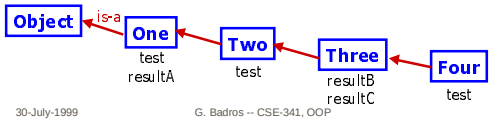
Example of super

One is-a Object
  test ^ 1
  resultA ^ self test

Three is-a Two
  resultB ^ self resultA
  resultC ^ **super** test

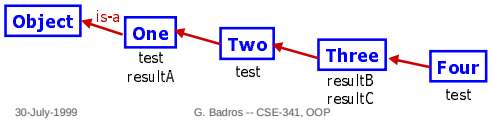
Two is-a One
  test ^ 2

Four is-a Three
  test ^ 4



Sample message sends

three test ⇒ 2
four resultA ⇒ 4
three resultB ⇒ 2
four resultB ⇒ 4
three resultC ⇒ 2
four resultC ⇒ 2



A small change

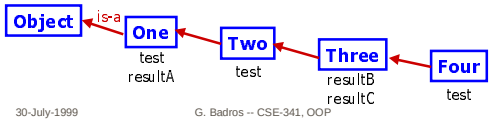
Added this method

One is-a Object
  test ^ 1
  resultA ^ self test

Two is-a One
  test ^ 2

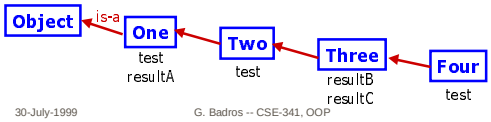
Three is-a Two
  test ^ 3
  resultB ^ self resultA
  resultC ^ **super** test

Four is-a Three
  test ^ 4



More message sends

three test ⇒ 3
four resultA ⇒ 4
three resultB ⇒ 3
four resultB ⇒ 4
three resultC ⇒ 2
four resultC ⇒ 2



Recursion

```
"Integer method"
factorial
  "Answer the factorial of the receiver."
  self > 1
      ifTrue: [^(self - 1) factorial * self].
  self < 0
      ifTrue: [^self error: 'negative factorial'].
  ^1
```


Conditionals

- Conditionals are implemented as messages sent to instances of Boolean
- Message selectors are...

```
ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:
```

- Arguments are ***blocks***

Blocks ⇔ Scheme's lambda

[i := i + 1]. ⇔ (lambda () (set! i (+ i 1)))

| i | [i := i + 1] ⇒ a HomeContext

| i b | i := 0. b := [i := i + 1].
  b value. b value. b value. i ⇒ 3

b value. ⇔ (b)

➪ **value** message evaluates the block

Blocks package up code

- Often used to suppress evaluation, as with their use with ifTrue:, ifFalse:, etc.
- Used in other control structures, too
- Simply anonymous procedures, with lexical scoping
- Can have arguments

Class Boolean

- True is-a Boolean
  true is-instance-of True
- False is-a Boolean
  false is-instance-of False

"True method"
  ifTrue: aBlock ^ aBlock value
"False method"
  ifTrue: aBlock ^ nil

Class Boolean has no state!

(0 = 0) ifTrue: [Transcript show: 'yes'; cr]

true, so execute True's method:
  ^ [Transcript show: 'yes'; cr] value

(0 = 1) ifTrue: [Transcript show: 'yes'; cr]

false, so execute False's method:
  ^ nil

While loops

"Context methods"
whileTrue: aBlock
whileFalse: aBlock
Example:

```
| j |   j:=10.
[ j > 0] whileTrue:
    [ Transcript show: j asString; cr. j:=j–1]
```

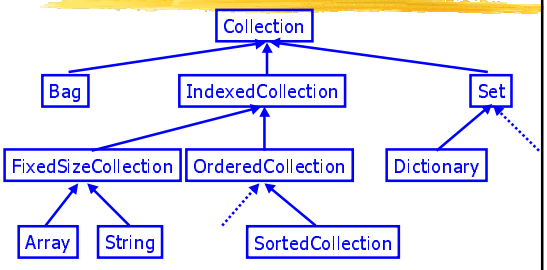

Common mistakes

[0 == 0] ifTrue: ['yes' printOn: Transcript]
Error! Message not understood

(0 == 1) ifTrue: ('yes' printOn: Transcript)
Always prints 'yes' to Transcript.

(j > 0) whileTrue: [j:=j–1]
Never terminates—only evaluates conditional once!

Collection classes



Collections understand lots of messages...

isEmpty	answers a boolean
includes: anObject	answers a boolean
add: anObject	adds the object
remove: anObject	removes the object
do: aBlock	like Scheme's for-each
collect: aBlock	like Scheme's map
select: aBlock	like Scheme's filter

do: and collect: messages

```
| a | a:=#(1 2 3) .
  a  do: [ :val | val printOn: Transcript ].
  Transcript cr

| a | a:=#(1 2 3) .
  a collect: [ :val | val + 1] ⇒ (2 3 4)
```

A block that takes an argument like Scheme's: (lambda (val) (+ val 1))

Dictionary class

```
| s | s := Dictionary new.
s at: #foo put: 19;
  at: #bar put: 37.
Array with: (s at: #bar) with: (s at #foo)
  ⇒ (37 19)
s asArray ⇒ (37 19)
s at: #baz ⇒ Error, absent key!
```


The Smalltalk object

- Smalltalk is-instance-of SystemDictionary
- Records all global objects
  - Smalltalk at: #Boolean
  - Smalltalk at: #Turtle
  - Smalltalk implementorsOf: #ifTrue:
- Can change the way Smalltalk itself works!

Smalltalk summary

- First dynamically-typed class-based object-oriented language
- Everything is an object—pure OO
- First class functions, lexical scoping, GC
- Programming environment is an important asset of the language
- Concreteness of objects was meant in part to be a pedagogical tool