Introduction to Object-oriented Programming in Smalltalk
Objects are responsible for their own actions!

• In procedural programming, I write code that reaches into the internals of some data structure and twiddles with the bits
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me

![Diagram of object interactions]

- String
- copy
- print
- concat
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me.
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me.
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me.
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me.
• In O-O programming, I politely request some other object to perform some work on my behalf, and it politely answers me.
Computation as Simulation

- Encapsulation is key
- Autonomous objects in the program represent objects in the real world
  - just like discreet event simulation
- Antropomorphize!
  - It’s OK to think about *this* object talking to *that* object…
  - in fact, it’s recommended
Programming Philosophy

• Object-Oriented programming is programming by simulation.
  – The algorithm is less important than the structure of the solution.

• When requirements change:
  – If the structure represented the structure of some ‘reality’, then the new requirements will be consistent in that reality.
  – Object-oriented design is the search for this structure: uncover the structure rather than construct in isolation.
Shopping vs. Building

• Constructing an Object-oriented application is a process of shopping for the components that one needs
  – occasionally we add a new item to the shelf.
  – usually we can find a component that *almost* fits.

• The *openness* of an OO language allows the programmer to change the component that *almost* fits into one that is a *good* fit.
  – works only if we have a rich set of components on the shelf, and if they are open to change.
Is this the *only* view of OO Programming?

No! People disagree on the meaning and role of:

1. Encapsulation
2. Types
3. Inheritance
4. Polymorphism
5. Sets and classes
Smalltalk

- Squeak is an open-source version of Smalltalk.
  - Smalltalk is still the best example of a Pure O-O language
  - The Squeak workspace is a place in which you can create and interact with objects.

- Large and active community of contributors
  - Runs “bit identical” on just about any platform, including many PDAs
The Squeak Environment

A “place” to experiment with objects

• Forget applications, files, compilers, data…
• Focus on objects
The Squeak World

- Embedded in a curve, then you can ask to have the text follow that curve, as illustrated in this image.
- You can also ask to have the text avoid occlusions, in which case it will do its best to avoid sibling submorphs that are in front of it.
- Kerning (cmd sh + or -) can help with the awkward spacing that results from narrow margins.
- Other morphs can also be embedded in text as glyphs.
- Kerning placement is based on the top left corner of the morph's bounding box.

Text in this rectangle flows around me.

Text morphs can be chained together, causing their contents to flow between containers as the contents or the containers change.

If a TextMorph is embedded in another morph, then you can ask to have it fill the space of that morph.
Smalltalk Syntax

• No syntax for classes, packages, etc.
  – Class creation and method categorization are done *imperatively* using the development tools

• The method syntax is simple, but different

```
>= aString
  "Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

^ (self compare: self with: aString collated: AsciiOrder) >= 2
```
Smalltalk Syntax

• No syntax for classes, packages, *etc.*
  
  – Class creation and method categorization are done *imperatively* using the development tools

• The method syntax is simple, but different

```smalltalk
>= aString

"Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

^ (self compare: self with: aString collated: AsciiOrder) >= 2
```

Method name
Smalltalk Syntax

• No syntax for classes, packages, etc.
  – Class creation and method categorization are done *imperatively* using the development tools

• The method syntax is simple, but different

```plaintext
>= aString

"Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

^ (self compare: self with: aString collated: AsciiOrder) >= 2
```

name of argument
**Smalltalk Syntax**

- No syntax for classes, packages, *etc.*
  - Class creation and method categorization are done *imperatively* using the development tools
- The method syntax is simple, but different

```smalltalk
>= aString

"Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

^ (self compare: self with: aString collated: AsciiOrder) >= 2
```

*method comment*
Smalltalk Syntax

- No syntax for classes, packages, etc.
  - Class creation and method categorization are done *imperatively* using the development tools

- The method syntax is simple, but different

```smalltalk
>= aString
"Answer whether the receiver sorts after or equal to aString. The collation order is simple ascii (with case differences)."

^ (self compare: self with: aString collated: AsciiOrder) >= 2
```

The code!
Read code

• The best way to become familiar with Smalltalk programming is to read the code in the image

• Expect to read 10 to 100 lines of code for each one that you write

  • If you find that you are writing long methods, you haven't “got it” yet.

• Find a method in the image that does something like what you want, and learn from it
# Smalltalk — The Language

## Literal Objects

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>The unique object 27</td>
</tr>
<tr>
<td>18.5</td>
<td>The floating point number 18.5</td>
</tr>
<tr>
<td>1.85e1</td>
<td>same as above</td>
</tr>
<tr>
<td>'a string'</td>
<td>a string</td>
</tr>
<tr>
<td>#request</td>
<td>the symbol request. It is unique; two symbols</td>
</tr>
<tr>
<td></td>
<td>with the same name denote the same object</td>
</tr>
<tr>
<td>$r</td>
<td>the single character r</td>
</tr>
<tr>
<td>#(3. 2.7 'a string')</td>
<td>an array literal. This is a heterogeneous array</td>
</tr>
</tbody>
</table>
Sending Messages

Unary Message (no arguments)

7 printString

- receiver (target of message)
- selector

- selector is a keyword-like symbol
  - examples: 3 factorial
  - 7 negated
  - $c$ asInteger
  - note: no colon at the end of the symbol
Binary Message (one argument!)

- selector is one or two special characters

$7 = 5$  \hspace{0.5cm} \text{message} = 5 \text{ sent to object 7}$

$i + 3$ \hspace{0.5cm} \text{message} + 3 \text{ sent to object } i$

$17 \div 3$ \hspace{0.5cm} \text{message} \div 3 \text{ sent to integer object 17 (result is 5)}$

$17 / 3$ \hspace{0.5cm} \text{message} / 3 \text{ sent to integer object 17 (result is )}
Binary Message (one argument!)

- selector is one or two special characters

7 = 5 \quad message = 5 \quad sent to object 7
i + 3 \quad message + 3 \quad sent to object \ i
17 // 3 \quad message // 3 \quad sent to integer object 17
\quad (result is 5)
17 / 3 \quad message / 3 \quad sent to integer object 17
\quad (result is )

Not exactly; \ i \ is not an object. It's a variable that's bound to an object
Keyword Messages

• one or more arguments
  – Examples:
    
    #(3 5 7 9 11) at: 2
    game movefrom: pinA to: pinB using: pinC
    5 between: 0 and: 9

• The colon ‘:’ indicates to the parser that an argument follows the keyword.
Order of Evaluation

• The receiver (or an argument) can be another invocation (message expression)

• Evaluation order is
  – parenthesed invocations
  – unary invocation, evaluated left to right
  – binary invocations, evaluated left to right
  – keyword invocations

• No “priorities” for particular operators
  – * does not bind more tightly than +
Cascaded Messages (syntactic sugar)

anArray at: 1 put: 9.
anArray at: 2 put: 11.
anArray at: 3 put: 13.

• This can be abbreviated as

anArray at: 1 put: 9; at: 2 put: 11; at: 3 put: 13

receiver for all 3 messages

“receiverless messages”

• Result is that of the last message send

Transcript show: 'Hello World'; cr
Variables

Instance Variables

• The names of the “slots” in an object, which make up its representation.

• declared in the class

  `instanceVariableNames: 'name1  name2'

Temporaries

• Names local to a method body or block

  `student  professorAtOGI`
Assignment

\[
x := 3 + 5
\]

- make \(x\) name the object resulting from the evaluation of the expression \(3 + 5\)

\[
y := \text{Array new: 1000000}
\]

- make \(y\) name a new 1MB array

- Variables name objects

- They do not provide storage for objects

- Assigning to a variable makes it name a different object

- no object is created or copied by assignment
Learning More

• Finding Classes
  – By name or fragment of a name
    - command-f in the Class-category pane of a browser
  – By selecting a morph and choosing browse morph class from the debug menu
• Finding methods
  
  – By name fragment or by example — with the method finder
  
  – Smalltalk browseMethodsWhoseNamesContain: 'screen'
  
  – Smalltalk browseMethodsWithString: 'useful', or highlight the string and type command-E
  
  – highlight a selector, choose implementors of … (command-m) or senders of … (command-n)
Finding Answers

Some invaluable resources:

• The Squeak “Swiki”
  – a wiki is a website where anyone is free to contribute to editing and maintenance
  – http://minnow.cc.gatech.edu/squeak
  - snapshot at http://swikimirror.squeakspace.com/

• Squeak.org
  – Documentation, tutorials, swikis, other sites, books and papers, downloads, and information on …
• The Squeak mailing list
  – a friendly place where “newbies” are made welcome
  – squeak-request@cs.uiuc.edu
  – Archive of [FIX]es, [ENH]ancements, [GOODIE]s…
    http://swiki.gsug.org:8080/SQFIXES
  – Searchable archive of whole list
    http://groups.yahoo.com/group/squeak
Creating Objects in Smalltalk

• Object are created by sending a message to some other (existing!) object called a **factory**

• *Usually*, the factory object is a class, e.g.

  ```smalltalk
  OrderedCollection new.
  Array with: 'one' with: 'two' with: 'three'.
  s := Bag new.
  ```

• The object will be deallocated automatically when it's no longer needed (garbage collected)
Blocks

- Blocks are Smalltalk objects that represent Smalltalk code
  \[ [1 + 2] \]

They can have arguments:

\[ [:x | 1 + x] \]

*compare with* \( \lambda x. 1 + x \)

- Blocks understand messages in the value family:
  
  | value | value: | value: value: | value: value: value: |

- The Block is *not* evaluated until it receives a *value* message
Examples of Blocks

• If-then-else is not a built-in control structure: it’s a message

  aBoolean ifTrue: trueBlock ifFalse: falseBlock

  discountRate := (transactionValue > 100)
  ifFalse: [0.05] ifTrue: [0.10]

• You can build your own control structures:

  ( keyEvent controlKeyPressed )
  and: [keyEvent shiftKeyPressed]
Returning an Answer

↑ returns an answer from a method

- if there is no ↑, the method returns self
- ↑ is very useful to return from a block

```small
color
  color ifNil: [↑ Color black].
  ↑ color
```

- ↑ in a block returns from the method in which the block is defined
- not the method that evaluates the block!
Arrays

• Arrays in Smalltalk are Objects

• They are “special” in 2 ways

  1. there is language syntax to create them

    #(1 3.4 #symbol)  an array literal

    {4-3. 17/5 asFloat . ('sym','bol') asSymbol}

      a dynamically constructed array

    Array with: 4-3 with: 17.0/5 with: #symbol  the same

  2. there are ByteArrays, FloatArrays as well as Arrays
Characters & Strings

• Characters are also objects
  $H$ is the literal for the character H
  $H$ asciiValue is 72
  $H$ digitValue is 17, $3$ digitValue is 3

• **collect:** creates a new array by applying a function to all elements of the receiver

  '01234567890ABCDEF' asArray
  collect: [ :each | each digitValue]
  evaluates to #(0 1 2 3 4 5 6 7 8 9 0 10 11 12 13 14 15)
Other enumeration methods

anArray **do:** aBlock

applies aBlock to each element of anArray, and answers anArray

anArray **withIndexCollect:** a2ArgumentBlock

answers the new array containing the results of applying a2ArgumentBlock to each element of anArray, together with its index.

anArray **withIndexDo:** a2ArgumentBlock
Examples

#(one two three four) withIndexCollect:
[ :each :i |
  each,' = ', i asString]

evaluates to #('one = 1' 'two = 2' 'three = 3' 'four = 4')

#(one two three four) withIndexDo:
[ :each :i |
  Transcript nextPutAll: each,' = '; show: i; cr]

evaluates to #(#one #two #three #four), i.e., the receiver
Indexing Arrays

• {#eins. #zwei. #drei} at: 1
• {#eins. #zwei. #drei} first
• {#eins. #zwei. #drei} third
• {#eins. #zwei. #drei} at: 2 put: #deux

 modifies the receiver, and answers #deux
Assignment 1: Whole objects

- Parse numerals into numbers without using explicit loops or recursion
- Use the algorithm shown