The Smalltalk Environment, SUnit, and Inheritance

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:
  ```smalltalk
  [ 1 + 2 ]
  ```
- They can have arguments:
  ```smalltalk
  [:x | 1 + x]
  ```
- Blocks understand messages in the value family:
  ```smalltalk
  value
  value: value:
  value: value: value:
  ```
- The Block is not evaluated until it receives a value message.
- Blocks are not evaluated until they receive a value message.

Examples of Blocks

- If-then-else is not a built-in control structure: it’s a message
  ```smalltalk
  aBoolean ifTrue: trueBlock ifFalse: falseBlock
  ```
- You can build your own control structures:
  ```smalltalk
  (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.
  ```smalltalk
  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.

The Smalltalk Collections

Q: What is a Collection?
A: An object that understands (some of) the following methods:

- isEmpty
- size
- includes:
- occurrencesOf:
- do:
- select:
- collect:
- reject:
- detect:
- detect:ifNone:
- inject:into:

Q: What is a Collection?
A: An object that understands (some of) the following methods:
Collections (cont.)

Q: Which classes have these methods?
A: Lots! In particular, most subclasses of Collection

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Bag</td>
</tr>
<tr>
<td>Interval</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Array</td>
<td>SortedCollection</td>
</tr>
<tr>
<td>Ordered Collection</td>
<td>LinkedList</td>
</tr>
<tr>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

What’s the Difference?

Each of these classes has some interesting refinement of the basic protocol

- Indexed Collections  
  - map an index to a value with at: (also at: put:)
- Extensible Collections  
  - size can be changed with add: (and remove:)
- Sequenceable Collections  
  - Indexed Collections on which we can sequence through the index set; supports first:, do:, collect: …

• Ordered Collections
  - access, insertion and removal based on the order are allowed: after: before: add:before: add:beforeIndex:

• Sorted Collections
  - The order is maintained by a relation (block) supplied explicitly with sortBlock: . at:put: is not understood.

If we regard these classes as a way of specifying interfaces (aka protocols) we can arrange them in a lattice by inclusion.

Abstract Classes in Smalltalk

Smalltalk classes are sometimes used to group behavior that is not complete enough to build an object! Such classes are called:

• abstract classes, or abstract superclasses
  - collection>>add: newObject
    "include newObject as one of my elements.
    Answer newObject:"
    self subclassResponsibility
  - collection>>addAll: aCollection
    aCollection do: [:each | self add: each].
    ↑ aCollection

Inheritance in Action!

• Subclasses of Collection don’t need to implement addAll:
  - it will be “inherited”
  - it will work if and only if they implement add:
• Partially abstract superclasses are a convenient place to put common code
• It can be hard to know if a class is abstract or concrete
  - Hint: try sending new or new: to the class
Arrays

- Arrays in Smalltalk are Objects
  - Array is a subclass of Collection
  - Arrays are “special” in 2 ways
    1. there is language syntax to create them
      ```smalltalk
      #(1 3.4 #thing)
      ```
      an array literal
      ```smalltalk
      (4-3, 17.5 asFloat, {'thi','ng'} asSymbol)
      ```
      a dynamically constructed array
      ```smalltalk
      Array with: 4-3 with: 17.0/5 with: #symbol
      ```
    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)`

- collect: is part of the enumeration protocol

Examples

```smalltalk
#(one #two #three #four) withIndexCollect:
[ :each :i | each,' = ', i asString]
evaluates to #('one = 1' 'two = 2' 'three = 3' 'four = 4')
```

```smalltalk
#(one #two #three #four) withIndexDo:
[ :each :i | Transcript nextPutAll: each,' = '; show: i; cr]
evaluates to #(#one #two #three #four), i.e., the receiver
```

Other enumeration methods

```smalltalk
anArray do: aBlock
``` applies aBlock to each element of anArray, and answers anArray

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

```smalltalk
anArray withIndexDo: a2ArgumentBlock
``` modifies the receiver, and answers #deux

Indexing Arrays

- `{#eins. #zwei. #drei}` at: 1
- `{#eins. #zwei. #drei}` first
- `{#eins. #zwei. #drei}` third
- `{#eins. #zwei. #drei}` at: 2 put: #deux
Names

• Names are the primary means of communication
  • Smalltalkers are fanatic about good names

• Capitalization conventions
  • local variables start with a lower-case letter
  • non-locals start with an upper-case letter
  • new words are capitalized
    • pairwise + product => pairwiseProduct
    • with + all + subclasses => withAllSubclasses

Naming Guidelines

• Name methods after what they accomplish
  • … not after the mechanism used in the implementation
  • imagine a very different implementation.
    • could you name this imagined method the same?
  • Use the same name as the method in the other class that does a similar thing

Example

• what’s the meaning of
  aSwitch on, or
  aSwitch setState: true ?

• What about:
  aSwitch isOn
  aSwitch turnOn
  aSwitch toggle ?

Naming Guidelines

• Name variables after their roles
  • instance variables and temporary variables should be named after their role
    • sum
    • result
    • bounds
    • don’t add a temporary variables unless there is a reason to do so!
    • b := self bounds.
      children do: [:each | … b topLeft … b bottomRight …]

Unit Testing

• Code that isn’t tested doesn’t work
  • Well, it’s true of my code — with the exception of simple accessor

• Two kinds of testing
  • Unit testing
  • Functional testing

What are tests for?

• Tests are an executable specification of the functionality that they cover
  • always synchronized with the code

• Tests increase the likelihood that the code is correct.
  • When you introduce a bug, you are more likely to find it very quickly, while it is still easy to fix

• Writing “tests first” improves your interfaces
  • you see your code from the client’s point of view

• The presence of tests gives you the courage to make structural changes to the code; refactoring
  • refactoring is essential to prevent creeping entropy
Test-driven Development

- When creating fresh code:
  - First write a test
  - only then write the code that makes the test run
- When maintaining old code
  - First write a (failing) test to isolate the bug
  - then fix the bug
  - … and run the whole test suite

Assignment 1: Whole objects

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

Where to put the Parsing Methods

- Where should the methods go in the class hierarchy?

<table>
<thead>
<tr>
<th>parseAsNumeral</th>
<th>digitValue</th>
<th>reversed</th>
<th>pairedWithPowersOf10</th>
<th>pairwiseProduct</th>
<th>sum</th>
</tr>
</thead>
</table>

Grading Rubric

<table>
<thead>
<tr>
<th>Name: CSE 245 Advanced Programming</th>
<th>Assignment 1: parse numerals</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Equipment</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td></td>
</tr>
</tbody>
</table>

Portland State University

CSE 245: Advanced Programming