CSCI-344
Programming Language Concepts (Section 3)

Lecture 20

Smalltalk Collections

Instructor: Hossein Hojjat

October 24, 2016

Where we are

Done:

- Smalltalk: an object-oriented language
- Inheritance
- Blocks: control structures

This session:

- Smalltalk Collections, Double Dispatch

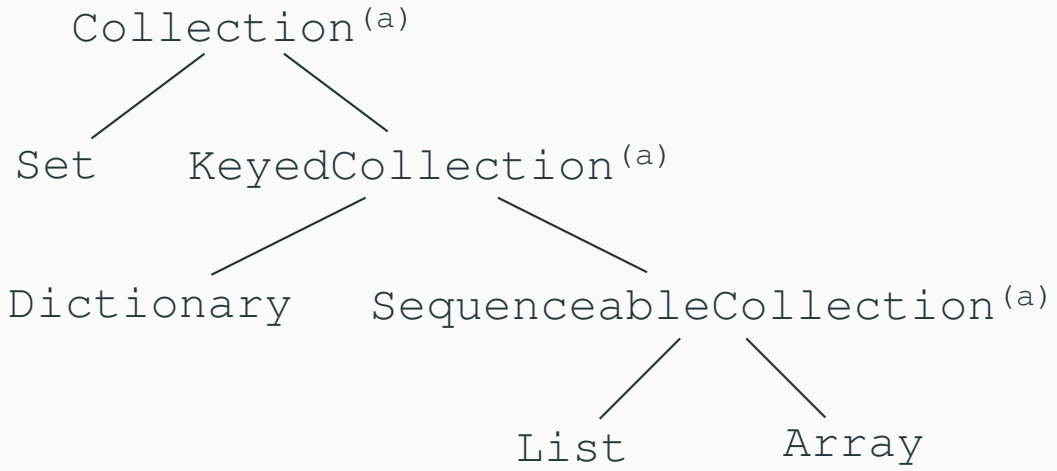
Collections

- Smalltalk itself is small, the initial basis contains most of the magic!
- Smalltalk blue book has around 90 pages on language,
  300 pages on library
  - Smalltalk-80: The Language and its Implementation
- Collections: objects that contain a group of other objects
- Top-level collection class is `Collection`
  - Abstract superclass that defines generic behavior
    common to collections
  - For example: `add:` , `isEmpty` , ...

Collections: Example

```
-> (val li (new List))
List( )
-> (val i 0)
0
-> (whileTrue: {(< i 10)} {(add: li i) (set i (+ i 1))})
nil
-> (includes: li 5)
<True>
-> (remove: li 4)
4
-> (size li)
9
-> (remove: li 10)
run-time error: tried-to-remove-absent-object
-> (do: li (block (x) (print x) (print #,)))
0,1,2,3,5,6,7,8,9,nil
```

Collections: Class Hierarchy



`Collection`	contain things
`Set`	objects in no particular order
`KeyedCollection`	objects accessible by keys
`Dictionary`	any key
`SequenceableCollection`	keys are consecutive integers
`List`	can grow and shrink
`Array`	fixed size, fast access

Collection Protocol: Mutators

`add: newObject`	Add argument
`addAll: aCollection`	Add every element of arg
`remove: oldObject`	Remove arg, error if absent
`remove:ifAbsent: oldObject exnBlock`	Remove the argument, evaluate `exnBlock` if absent
`removeAll: aCollection`	Remove every element of arg

Exercise

What is the output of the following code?

```
(val i 0)
(val s1 (new Set))
(val s2 (new Set))
(whileFalse: {(= (size s1) 10)}
             {(add: s1 i)(set i (+ i 1))})
(set i 0)
(whileTrue: {(not (= (size s2) 5))}
            {(add: s2 i)(set i (+ i 2))})
(removeAll: s1 s2)
(print s1)
```

Exercise

What is the output of the following code?

```
(val i 0)
(val s1 (new Set))
(val s2 (new Set))
(whileFalse: {(= (size s1) 10)}
             {(add: s1 i)(set i (+ i 1))})
(set i 0)
(whileTrue: {(not (= (size s2) 5))}
            {(add: s2 i)(set i (+ i 2))})
(removeAll: s1 s2)
(print s1)
```

Answer. `Set( 1 3 5 7 9 )`

Collection Protocol: Observers

`isEmpty`	Is it empty?
`size`	How many elements?
`includes: anObject`	Does receiver contain arg?
`occurrencesOf: anObject`	How many times?
`detect: aBlock`	Find and answer element satisfying `aBlock` (cf μScheme exists?)
`detect:ifNone: aBlock exnBlock`	Detect, recover if none
`asSet`	Set of receiver's elements

Exercise

What is the output of the following code?

```
(val i 0)
(val li (new List))
(whileFalse: {(= (size li) 10)}
             {(add: li i)(set i (+ i 5))})
(detect:ifNone: li (block (x) (= (mod: x 7) 0))
                   {(add: li 50)} )
(includes: li 50)
```

Exercise

What is the output of the following code?

```
(val i 0)
(val li (new List))
(whileFalse: {(= (size li) 10)}
             {(add: li i)(set i (+ i 5))})
(detect:ifNone: li (block (x) (= (mod: x 7) 0))
                   {(add: li 50)} )
(includes: li 50)
```

Answer. `<False>`

Collection Protocol: Iterators

`do: aBlock`	For each element `x`, evaluate `(value aBlock x)`
`inject:into: thisValue binaryBlock`	Essentially μScheme `foldl`
`select: aBlock`	Essentially μScheme filter
`reject: aBlock`	Filter for *not* satisfying `aBlock`
`collect: aBlock`	Essentially μScheme map

Exercise

What is the output of the following code?

```
(val i 0)
(val l1 (new List))
(whileTrue: {(<= (size l1) 10)}
            {(add: l1 i)(set i (+ i 1))})
(val l2 (reject: l1 (block (x) (= (mod: x 2) 1))))
(inject:into: l2 0 (block (x y) (+ x y)))
```

Exercise

What is the output of the following code?

```
(val i 0)
(val l1 (new List))
(whileTrue: {(<= (size l1) 10)}
            {(add: l1 i)(set i (+ i 1))})
(val l2 (reject: l1 (block (x) (= (mod: x 2) 1))))
(inject:into: l2 0 (block (x y) (+ x y)))
```

Answer. 30

Implementing Collections

```
(class Collection Object
  () ; abstract
  (method do:  (aBlock)
    (subclassResponsibility self))
  (method add: (newObject)
    (subclassResponsibility self))
  (method remove:ifAbsent (oldObj exnBlock)
    (subclassResponsibility self))
  (method species ()
    (subclassResponsibility self))
  <other methods of class Collection>
)
```

Reusable methods

These methods always work Subclasses can override (redefine) with more efficient versions

```
<other methods of class Collection>=
(method addAll: (aCollection)
  (do: aCollection (block(x) (add: self x)))
  aCollection)
(method size () (locals temp)
  (set temp 0)
  (do: self (block(_) (set temp (+ temp 1))))
  temp)
```

`species` Method

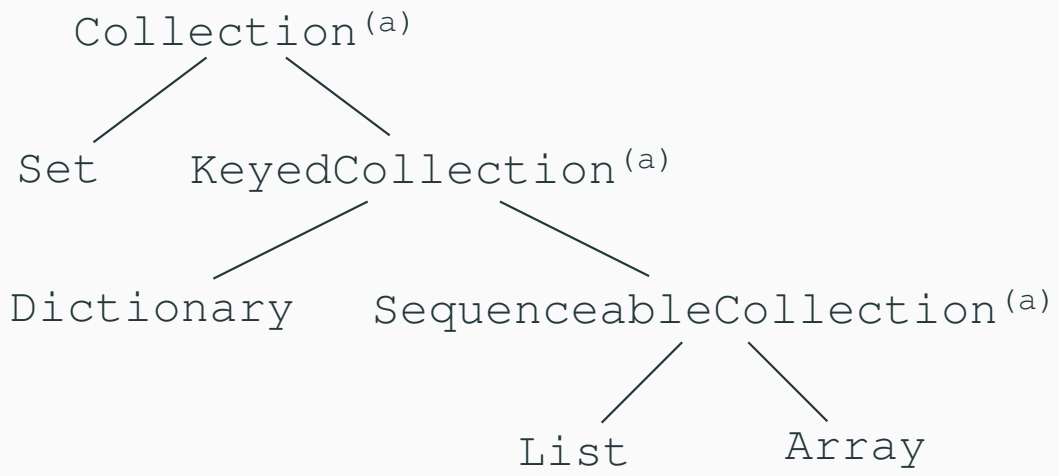
- Create "collection like the receiver"
- Example: filtering

```
(method select: (aBlock) (locals temp)
  (set temp (new (species self)))
  (do: self (block (x) (ifTrue: (value aBlock x)
                                {(add: temp x)})))
temp)
```

Collections: Class Hierarchy

- Four crucial methods of Collection

  `do: , add: , remove:ifAbsent , species`



Collections: Class Hierarchy

- Four crucial methods of Collection

  `do:` , `add:` , `remove:ifAbsent` , `species`

```
-> (val i 1)
(val a (new: Array 10))
(whileTrue: {(<= i 10)}
   {(at:put: a i i) (set i (+ i 1))})
-> a
( 1 2 3 4 5 6 7 8 9 10 )
-> (add: a 11)
run-time error: arrays-have-fixed-size
```

Collections: Class Hierarchy

- Four crucial methods of Collection

  `do:` , `add:` , `remove:ifAbsent` , `species`

- Superclass `Collection` defines method `add:`
  but subclass `Array` fails to implement it!
- Deep class hierarchy: when you have several layers of subclasses
- It can be hard to design deep class hierarchies
- It's good to share methods, but sharing too much is harmful!

Double Dispatch Pattern

- **Method dispatch:** decide which method should be invoked in response to a message
- **Single dispatch:** method to be invoked depends only on receiver of message
- **Double dispatch:** method to be invoked depends both on receiver and argument of message

Double Dispatch Pattern

Example: Addition

- Three classes for numbers: `Integer`, `Float`, `Fraction`
- Result type for addition operation does not only depend on receiver

```
-> (val num 5)
5
-> (val frc (num:den: Fraction 4 5))
4/5
-> (val flt (mant:exp: Float 3 4))
3x10^4
```

Receiver Class:

	Argument Class: **Integer**	**Fraction**	**Float**
Integer	Integer	Fraction	Float
Fraction	Fraction	Fraction	Float
Float	Float	Float	Float

Type of `(+ num1 num2)`

Double Dispatch Pattern

Solution

- Rather than implementing messages directly,
  method body sends message back to the argument
- Correct final method body can depend on both receiver and argument

- Method name encodes both operation
  and type of argument
  - `addIntegerTo:` , `addFloatTo:`
- In class Float:

```
(method + (anInteger) (addFloatTo:  anInteger self))
```

- In class Integer: `(method addFloatTo:  (aFloat)`

```
    (< code to add an Integer and a Float > )
```

Double Dispatch Pattern

- Consider evaluation of: `(+ 5.5 3)`
- Sends `+` message to object `5.5` with argument `3`
- Sends `addFloatTo:` message to `3` with argument `5.5`
- Code to add an `Integer` and a `Float` is executed

Summary

This Lecture

- Smalltalk Collections, Double Dispatch

Next Lecture

- Prolog