Reading

Smalltalk by Example, Alex Sharp,
Chapter 11 Collections
(N) - number of methods defined the class

Bold - commonly used classes
Overview

Array
Fixed size
Elements indexed by integers

Bag
No order or indexing
Repeats allowed

Dictionary
Hash table
Elements indexed by any object

Interval
Finite arithmetic progression

OrderedCollection
Growable array

Set
No order, indexing or repeats

SortedCollection
Sorted growable array

String
Fixed size array of characters

Symbol
String with unique instances

Text
Text that supports fonts, bold etc.
Arrays

Similar to arrays in other languages
Once created can not grow

Creating an Array

a := #( 1 3 5 7 6 4 2 ).        "A literal array"
b := Array with: 5 with: 9.    "Create an array with two elements"
c := Array new: 10        "Create array with 10 elements, all nil"

Accessing Elements

secondElement := a at: 2.      "indexing starts at 1"
firstElement := a first.
lastElement := a last.

a                "set first element to 12"
    at: 1
    put: 12.
Some Array Operations

numberOfElements := a size.
locationOfFiveInArray := a indexOf: 5.

jointList := a , b.     "Concatenation"
sublist := a
    copyFrom: 2
to: 4.


location := a
    indexOfSubCollection: #( 6 4)
    startingAt: 2
    ifAbsent: [-1].

More Array Operations

```plaintext
a
  replaceAll: 3
  with: 12.

a occurrencesOf: 2.

(a includes: 2) ifTrue: [blah].
(a contains: [:each | each odd] ) ifTrue; [ blah].

(a anySatisfy: [:each | each odd]) ifTrue: [blah].
(a allSatisfy: [:each | each odd]) ifTrue: [blah].

a isEmpty ifTrue; [blah].
```
OrderedCollections

A growable array
When add elements, OrderedCollections grows if needed

Like Java's Vector or ArrayList

Creating an OrderedCollection

a := #( 1 3 5 7 6 4 2 ) asOrderedCollection.

b := OrderedCollection new.

c := OrderedCollection with: 5 with: 9.

d := OrderedCollection new: 10.
OrderedCollection Methods

b  "Add elements grow if needed"
   add: 2;
   add: 5.

secondElement := a at: 2.
firstElement := a first.

a
   at: 1
   put: 12.

jointList := a , b.


fiveIndex := a indexOf: 5.

a
   removeAll: 3
   with: 12.

finumberOfElements := a size.

a remove: 5
Size, Capacity & Growing

Size - number of elements in collection

Capacity - number of elements collection can hold without growing

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
</table>
a := OrderedCollection new.
a size.  "Answers 0"
a capacity  "Answers 5"

6 timesRepeat: [a add: 'cat']

a size.  "Answers 6"

a capacity.  "Answers 10"
Dictionary

A hash table, like Java's Hashtable or HashMap
In arrays and ordered collections indexes are integers
In dictionaries indexes can be any object

| phoneNumbers |

phoneNumbers := Dictionary new.
phoneNumbers
  at: 'whitney'
  put: '594-3535'.

phoneNumbers
  at: 'beck'
  put: '594-6807'.

phoneNumbers
  at: 'donald'
  put: '594-7248'.

phoneNumbers at: 'donald'       "Returns '594-7248' "

phoneNumbers
  at: 'sam'
  ifAbsent: ['Not found'].

hash & =

Both hash and = are used to add/find elements in a dictionary

Hash determine where to start looking

= is used to separate items with the same hash value

If you redefine = in a class make sure that

if $a = b$ then $a$ hash = $b$ hash
Strings & Symbols

A String is an array of characters
Characters can be any Unicode character

Symbols are strings that are represented uniquely
  #ASymbol
  #'CanUseSingleQuotes'
  #cat

There is only one copy of a symbol with a given sequence of characters in the image

'cat' = 'cat'     "true"
'cat' == 'cat'    "false"

#cat = #cat       "true"
#cat == #cat      "true"
## Collection Creation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array new: 5</td>
<td>#(nil nil nil nil nil)</td>
</tr>
<tr>
<td>OrderedCollection new: 5</td>
<td>OrderedCollection()</td>
</tr>
<tr>
<td>Array with: 2 with: 1</td>
<td>#(2 1)</td>
</tr>
<tr>
<td>Bag with: 1 with: 1 with: 2</td>
<td>Bag(1 1 2)</td>
</tr>
<tr>
<td>Set with: 1 with: 1 with: 2</td>
<td>Set(1 2 )</td>
</tr>
</tbody>
</table>
### Converting

<table>
<thead>
<tr>
<th>asArray</th>
<th>cat' asSortedCollection</th>
</tr>
</thead>
<tbody>
<tr>
<td>asBag</td>
<td>SortedCollection ($a &quot;16r0061&quot; $c &quot;16r0063&quot; $t &quot;16r0074&quot;)</td>
</tr>
<tr>
<td>asSet</td>
<td>#( 3 9 1 4 ) asSortedCollection</td>
</tr>
<tr>
<td>asOrderedCollection</td>
<td>SortedCollection(1 3 4 9)</td>
</tr>
<tr>
<td>asSortedCollection</td>
<td>#( 1 2 3 2 1) asBag</td>
</tr>
<tr>
<td>asSortedCollection: aBlock</td>
<td>Bag(1 1 2 2 3)</td>
</tr>
</tbody>
</table>
## Sorting

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#( 3 9 1 4 ) asSortedCollection: [:x :y</td>
<td>x &gt; y ]</td>
</tr>
<tr>
<td>#( 3 9 1 4 ) asSortedCollection: [:x :y</td>
<td>x &lt; y ]</td>
</tr>
<tr>
<td>#( 3 9 1 4 ) asSortedCollection</td>
<td>SortedCollection(1 3 4 9)</td>
</tr>
<tr>
<td>#( 'dog' 'mat' 'bee' ) asSortedCollection</td>
<td>SortedCollection('bee' 'dog' 'mat')</td>
</tr>
<tr>
<td>#( 'dog' 'mat' 'bee' ) asSortedCollection: [:x :y</td>
<td>(x at: 2) &lt; (y at: 2)]</td>
</tr>
<tr>
<td>#( 'dog' 'mat' 'bee' ) asSortedCollection: [:x :y</td>
<td>(x at: 2) &gt; (y at: 2)]</td>
</tr>
</tbody>
</table>
Common Methods

size
capacity
at: indexOrKey
at: indexOrKey put: anElement

add: anElement
addAll: aCollection

remove: anElement
remove: anElement ifAbsent: aBlock
removeAll: aCollection

isEmpty
isNil
includes: anElement
occurrencesOf: anElement
Enumerating

| sum |
sum := 0.
#( 3 2 1) do: [:each | sum := sum + each squared].
^sum

Verses

| data sum |
data := #( 3 2 1).
sum := 0.
1 to: data size do: [:each | sum := sum + (data at: each) squared].
^sum
## Common Enumerations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this is an example'</code></td>
<td>select: `[:each</td>
<td>each isVowel ]`</td>
</tr>
<tr>
<td><code>#(1 5 2 3 6)</code></td>
<td>reject: `[:each</td>
<td>each even ]`</td>
</tr>
<tr>
<td><code>#(1 2 3 4 5)</code></td>
<td>collect: `[:each</td>
<td>each squared ]`.</td>
</tr>
<tr>
<td><code>'hi mom'</code></td>
<td>collect: `[:each</td>
<td>each asUppercase ]`.</td>
</tr>
<tr>
<td><code>#(1 7 2 3 9 3 50)</code></td>
<td>detect: `[:each</td>
<td>each &gt; 8 ]`</td>
</tr>
</tbody>
</table>
### Brief Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>do: aBlock</td>
<td>Evaluate aBlock with receiver's elements</td>
</tr>
<tr>
<td>select: aBlock</td>
<td>Return elements that make aBlock true</td>
</tr>
<tr>
<td>reject: aBlock</td>
<td>Return elements that make aBlock false</td>
</tr>
<tr>
<td>collect: aBlock</td>
<td>Return result of evaluating aBlock on each element</td>
</tr>
<tr>
<td>detect: aBlock</td>
<td>Return first element that makes aBlock true</td>
</tr>
<tr>
<td>inject: initialValue into: binaryBlock</td>
<td>Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value of the argument, thisValue, and the receiver as block arguments.</td>
</tr>
</tbody>
</table>
Inject:into: Example

Transcript
  clear;
  show: 'Partial';
  tab;
  show: 'Number';
cr.

#( 1 2 3 4 5) inject: 0 into:
  [:partialSum :number |
  Transcript
    show: partialSum;
    tab;
    show: number;
    cr.
  partialSum + number.]

<table>
<thead>
<tr>
<th>Partial</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

Result: 15
More Examples

Product

#( 1 2 3 4)
  inject: 1
  into: [:partialProduct :number | partialProduct * number]

Vowel Count

'hi mom' inject: 0 into:
  [:partial :each |
    each isVowel
    ifTrue:[partial + 1]
    ifFalse:[partial]]
'this is an example' keysAndValuesDo:
 [:key :value |
 value isVowel
 ifTrue:
   [Transcript
    show: key;
    tab;
    show: value;
    cr]]
Some Fun

Transcript
  show: 'Digit';
  tab;
  show: 'Frequency';
  cr.
100 factorial asString asBag sortedElements do:
  [:each |
    Transcript
      show: each key;
      tab;
      show: each value;
      cr]

<table>
<thead>
<tr>
<th>Digit</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
</tr>
</tbody>
</table>
Useful Enumerations on Sequenceable Collections

with:do:

| pairwiseSum |
pairwiseSum := OrderedCollection new.

#(1 3 5 7 9)
  with: #(2 4 6 8 10)
  do: [:first :second | pairwiseSum add: first + second].

^pairwiseSum

do:seperatedBy:

#(2 4 6 8)
  do: [:each | Transcript print: each ]
  separatedBy: [Transcript show: ',', ']

Transcript
  cr;
  flush

OrderedCollection (3 7 11 15 19)

2, 4, 6, 8
# fold:

| #(1 2 3) fold: [:a : b | a + b] | 6 |
|---------------------------------|---|
| #( 'A' 'cat' 'in' 'the' 'hat' ) fold: [:a : b | a , ' ' , b] | 'A cat in the hat' |

Evaluate a block with the 1st and the 2nd element of the receiver, then with the result of the first evaluation and the 3rd element, etc.
# piecesCutWhere:do:

<table>
<thead>
<tr>
<th>A sentence. Another sentence... Yet another sentence.</th>
</tr>
</thead>
<tbody>
<tr>
<td>piecesCutWhere:</td>
</tr>
<tr>
<td>[:each :next</td>
</tr>
<tr>
<td>each = $. and: [next = Character space]]</td>
</tr>
<tr>
<td>do:</td>
</tr>
<tr>
<td>[:each</td>
</tr>
<tr>
<td>Transcript</td>
</tr>
<tr>
<td>show: each printString;</td>
</tr>
<tr>
<td>cr]</td>
</tr>
</tbody>
</table>

| A sentence.'                                         |
| 'Another sentence...'                                 |
| 'Yet another sentence.'                               |

| #( 1 3 7 2 4 5 7 4 1 7 9)                             |
| piecesCutWhere:[:each :next | each > next]       |
| do: [:each | Transcript show: each printString; cr]        |

| #(1 3 7)                                             |
| #(2 4 5 7)                                           |
| #(4)                                                 |
| #(1 7 9)                                             |

# piecesCutWhere: block
- Indicates where to break receiver into pieces
- Does one character look ahead
- Character that cause break is the last element in the piece
runsFailing:do:

```ruby
#(1 3 7 2 4 5 7 4 1 7 9)
runsFailing:[:each | each = 7]
do:
    [:each |
    Transcript
    show: each printString;
    cr]
```

```ruby
#(1)
#(3)
#(2 4 5)
#(4 1)
#(9)
```

runsFailing: block
Determines where receiver is divided into pieces
Character that cause break is not in any piece

do: block is done on the pieces