CS 535 Object-Oriented Programming & Design
Fall Semester, 2001
Doc 8 Collections

Contents

Collections......................................................................................................................2
Creation ..........................................................................................................................4
  Special Array Creation..............................................................................................6
Converting ....................................................................................................................7
Accessing .....................................................................................................................11
Adding ..........................................................................................................................12
Removing ....................................................................................................................13
Testing ..........................................................................................................................15
Enumerating .............................................................................................................16
  do: .............................................................................................................................19
  select: aBlock ..........................................................................................................22
  reject: aBlock ..........................................................................................................22
  collect: aBlock .........................................................................................................23
  detect: aBlock ..........................................................................................................24
  inject: thisValue into: binaryBlock .........................................................................25

References


Reading

VisualWorks Application Developer’s Guide, Chapter 17 Collections

Or

Joy of Smalltalk, Ivan Tomek, Chapter 7

Copyright ©, All rights reserved.
2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Collections

Smalltalk has rich set of collections. Most of them should be familiar to Java programmers. We will cover some of the important collection classes.

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
More Collections

LinkedList
   A linked list

SharedQueue
   Used to pass data between processes

Symbol
   String with unique instances

Text
   Text that supports fonts, bold etc.
Common Collection Methods

Some methods may not be supported by all collection objects. There are a lot of methods not shown here.

Creation

Creation methods are sent to Collection classes

new
  Create a new instance of the receiver with no elements

new: anInteger
  Fixed size collections create a collection of size anInteger filled with default elements
  Variable sized collections create a collection with capacity anInteger, but no elements

with: anElement
  Create a new instance of the receiver with the given element

with: with:
with: with: with:
with: with: with: with:
  Create a new instance of the receiver with the given number of elements

withAll: aCollection
  Create a new instance of the receiver with each element of aCollection as an element in the new collection
## Creation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result printed</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>
<tr>
<td>Bag new</td>
<td>Bag()</td>
</tr>
<tr>
<td>OrderedCollection new</td>
<td>OrderedCollection()</td>
</tr>
</tbody>
</table>

### String new: 5
- Returns a String with 5 characters
- Each character has ASCII value 0

Note the results above are obtained by selecting one line of text at a time in a workspace and executing it with "print it"
Special Array Creation

Literal Array Creation

Format:

#( element1  element2 … elementN )

- Created at compile time
- All elements are treated as literals

Examples

#( 1 2 'cat' )

#( 1 123 54 45.3)

#( 'dog' 'mat' $c$ a $t$ )

What does not Work

| x |
---
x := 'test'.
#( x )

Since all elements in a literal array creation must be a literal, the value of x is not included in the array. The symbol for x is the element of the array.
Converting

asArray
asBag
asSet
asOrderedCollection
asSortedCollection
asSortedCollection: aBlock

Convert the receiver to the indicated collection

Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cat' asSortedCollection</td>
<td>SortedCollection ($a &quot;16r0061&quot; $c &quot;16r0063&quot; $t &quot;16r0074&quot;)</td>
</tr>
<tr>
<td>#( 3 9 1 4 ) asSortedCollection</td>
<td>SortedCollection(1 3 4 9)</td>
</tr>
<tr>
<td>#( 1 2 3 2 1) asBag</td>
<td>Bag(1 1 2 2 3)</td>
</tr>
<tr>
<td>'hi mom' asBag</td>
<td>Bag (Core.Character space $o &quot;16r006F&quot; $h &quot;16r0068&quot; $i &quot;16r0069&quot; $m &quot;16r006D&quot; $m &quot;16r006D&quot;)</td>
</tr>
</tbody>
</table>

Note $a printString returns '$a "16r0061"'. That is you get the character and its hex value. Very useful with whitespace characters, but can be annoying other times. I will edit these values out in some future slide to save space.
## 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>#( $2 $a $A $w) asSortedCollection</td>
<td>SortedCollection ($2 $A $a $w )</td>
</tr>
</tbody>
</table>

The block argument must return true when the first element precedes the second one

[:x :y | x < y ] is the Default Sort Block (increasing)
Sorting By Second Character

#( 'dog' 'mat' 'bee' ) asSortedCollection: [:x :y | (x at: 2) < (y at: 2)]

Result:

SortedCollection ('mat' 'bee' 'dog')
Mixing Elements

All elements in a sorted collection may be compared to any other element in the collection.

Each element must be comparable to the others in the collection.

The following results in a runtime error:

```bash
#( 1 'cat' $d) asSortedCollection
```
Accessing

size
Returns the current number of element in the collection

capacity
Returns the number of elements the collection could hold without growing

at: indexOrKey
Return the element stored at the index or key
Some collections want keys (Dictionary) some want indexes
Replaces standard array accessing a[k]

at: indexOrKey put: anElement
Store anElement at the index or key
Some collection wants keys (Dictionary) some want indexes

<table>
<thead>
<tr>
<th>collection</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection := #( 'a' 'b' 'c' 'd' ).</td>
<td></td>
</tr>
<tr>
<td>Transcript print: collection size.</td>
<td>4</td>
</tr>
<tr>
<td>Transcript print: collection capacity</td>
<td>4</td>
</tr>
</tbody>
</table>
| Transcript print: (collection at: 2). | 'b'
| Transcript print: (collection at: 1 put: 'cat'). | |
| Transcript show: collection printString. | '#("cat" "b" "c" "d")'

| collection := OrderedCollection new. | |
| Transcript print: collection capacity. | 10 |
| Transcript print: collection size | 0 |
Adding

Can not add to a fixed size collection like arrays or strings

Add methods return the element added to the collection

add: anElement
   Add anElement to the end of the receiver (a collection)

addAll: aCollection
   Add all elements of aCollection to the end of receiver

<table>
<thead>
<tr>
<th>a</th>
<th>Result on the transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := OrderedCollection with: $a.</td>
<td></td>
</tr>
<tr>
<td>Transcript show: a</td>
<td>OrderedCollection($a )</td>
</tr>
<tr>
<td>a add: 'cat'.</td>
<td></td>
</tr>
<tr>
<td>a add: 5.</td>
<td></td>
</tr>
<tr>
<td>Transcript show: a.</td>
<td>OrderedCollection($a &quot;cat&quot; 5)</td>
</tr>
<tr>
<td>a addAll: 'dog'.</td>
<td></td>
</tr>
<tr>
<td>Transcript show: a</td>
<td>OrderedCollection($a &quot;cat&quot; 5 $d $o $g)</td>
</tr>
</tbody>
</table>

Since 'dog' is a string, which is a collection, addAll: 'dog' adds the characters of 'dog' one at a time to the collection.
Removing

You can not remove from a fixed size collection like arrays or strings

remove: anElement
  Remove anElement from the receiver
  Throw an exception if anElement is not in the receiver

remove: anElement ifAbsent: aBlock
  Remove anElement from the receiver
  Execute aBlock if anElement is not in the receiver

removeAll: aCollection
  Remove all elements in aCollection from the receiver
  Throw an exception if any element of aCollection is not in the receiver
## Removing Examples

<table>
<thead>
<tr>
<th>data result original</th>
<th>Output in Transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>original := #( 4 3 2 1) asOrderedCollection.</td>
<td></td>
</tr>
</tbody>
</table>

| data := original copy. | |
| data remove: 3. | |
| Transcript show: data; cr. | OrderedCollection(4 2 1) |

| data := original copy. | |
| data remove: 5 ifAbsent: []. | |
| Transcript show: data; cr. | OrderedCollection(4 3 2 1) |

| data := original copy. | |
| data removeAll: #( 1 3). | |
| Transcript show: data; cr. | OrderedCollection(4 2) |

| result := data remove: 4. | |
| Transcript show: result; cr. | 4 |
| Transcript flush. | |
Testing

isEmpty
includes: anElement
occurrencesOf: anElement

Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#( 1 6) isEmpty</td>
<td>false</td>
</tr>
<tr>
<td>'cat' includes: $o</td>
<td>false</td>
</tr>
<tr>
<td>'mom' occurrencesOf: $m</td>
<td>2</td>
</tr>
<tr>
<td>#( 1 3 2 4 3) occurrencesOf: 3</td>
<td>2</td>
</tr>
</tbody>
</table>

Note the results above are obtained by selecting one line of text at a time in a workspace and executing it with "print it"
Enumerating

Enumeration:
Perform tasks on elements of a collection
Do not handle details of accessing each element

Some languages call this iteration

**Example - Sum of Squares**

```
| sum  |
sum := 0.
#( 1 7 2 3 9 3 50) do: [:each | sum := sum + each squared].
^sum
```

do: iterates or enumerates through the elements of the array

We could use a normal loop construct like:

```
| data sum |
data := #( 1 7 2 3 9 3 50).
sum := 0.
1 to: data size do: [:each | sum := sum + (data at: each) squared].
^sum
```
Loop Construct Verses Enumeration

The loop construct:
  Is more work
  Assumes the collection is ordered
  Will not work will bags, sets, and dictionaries

Enumeration is:
  Less work
  More general
  Just as fast

Use Enumeration over explicit loop constructs
## Basic Enumeration for all Collections

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do: aBlock</code></td>
<td>Evaluate <code>aBlock</code> with each of the receiver's elements as the argument.</td>
</tr>
<tr>
<td><code>select: aBlock</code></td>
<td>Evaluate <code>aBlock</code> with each of the receiver's elements as the argument. Collect into a new collection like the receiver, only those elements for which <code>aBlock</code> evaluates to true. Answer the new collection.</td>
</tr>
<tr>
<td><code>reject: aBlock</code></td>
<td>Evaluate <code>aBlock</code> with each of the receiver's elements as the argument. Collect into a new collection like the receiver only those elements for which <code>aBlock</code> evaluates to false. Answer the new collection.</td>
</tr>
<tr>
<td><code>collect: aBlock</code></td>
<td>Evaluate <code>aBlock</code> with each of the receiver's elements as the argument. Collect the resulting values into a collection like the receiver. Answer the new collection.</td>
</tr>
<tr>
<td><code>detect: aBlock</code></td>
<td>Evaluate <code>aBlock</code> with each of the receiver's elements as the argument. Answer the first element for which <code>aBlock</code> evaluates to true. Signal an Error if none are found.</td>
</tr>
<tr>
<td><code>inject: initialValue into: binaryBlock</code></td>
<td>Accumulate a running value associated with evaluating the argument, <code>binaryBlock</code>, with the current value of the argument, <code>thisValue</code>, and the receiver as block arguments.</td>
</tr>
</tbody>
</table>
do:
  do: aBlock
  Evaluate aBlock with each of the receiver's elements as the argument.

'this is an example' do:
  [:each |
    each isVowel ifTrue:[Transcript show: each]]

  Result in Transcript

iiæae
keysAndValuesDo: aBlock

Defined for keyed collections only (no bags & sets)

Sometimes one needs the element of a collection and the index of the element

'this is an example' keysAndValuesDo:
   [:key :value |
       value isVowel
       ifTrue:
           [Transcript
             show: key;
             tab;
             show: value;
             cr]]

Result in Transcript

3  i
6  i
9  a
12 e
14 a
18 e
Some Fun

Can you parse this program?
What does each message do?

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

Output In Transcript

<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>
**select: aBlock**

Return a new collection with the elements of the receiver that make the block evaluate to true

**Example**

```plaintext
| result |
result := 'this is an example' select: [:each | each isVowel ].
^result

Returned Value

'iiaeae'
```

**reject: aBlock**

Return a new collection with the elements of the receiver that make the block evaluate to false

**Example**

```plaintext
| result |
result := #( 1 5 2 3 6) reject: [:each | each even ].
^result

Returned Value

 #(1 5 3)```
collect: aBlock

Collects the return values of aBlock into new collection

Examples

| result |

result := #( 1 2 3 4 5) collect: [:each | each squared ].
^result

Returned Value

#(1 4 9 16 25)

| result |

result := 'hi mom' collect: [:each | each asUppercase ].
^result

Returned Value

'HI MOM'
detect: aBlock

Returns the first element in the receiver that makes aBlock evaluate to true

#( 1 7 2 3 9 3 50) detect: [:each | each > 8]

Returns

9
**inject: thisValue into: binaryBlock**

Accumulates a running value

inject:into is confusing the first time you see it.

**Compute Sum of Collection's Elements**

```
#( 1 2 3 4)
  inject: 0
  into: [:partialSum :number | partialSum + number]
```

**Compute Product of Collection's Elements**

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

**Count the Vowels in a String**

```
'hi mom' inject: 0 into:
  [:partial :each |
    each isVowel
    ifTrue:[partial + 1]
    ifFalse:[partial]]
```

Note the first two examples are used in Smalltalk code, there are easier ways to count vowels.
Detailed 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.]

Result in Transcript

<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>
Example - Computing Sum of Squares

C++ like Code

| data sum |
data := #( 1 7 2 3 9 3 50).
sum := 0.
1 to: data size do: [:each | sum := sum + (data at: each) squared].
sum

With do:

| sum |
sum := 0.
#( 1 7 2 3 9 3 50) do: [:each | sum := sum + each squared].
sum

With inject:into

#( 1 7 2 3 9 3 50) inject: 0 into: [:sum :each | sum + each squared]