CS 683 Emerging Technologies: Embracing Change  
Spring Semester, 2001  
Doc 5 Collections  
Contents  

Collections ................................................................. 2  
Creation ........................................................................ 4  
   Special Array Creation .............................................. 6  
Converting ........................................................................ 8  
Accessing ......................................................................... 9  
Adding ............................................................................... 10  
Removing .......................................................................... 11  
Testing ............................................................................. 13  
Enumerating ....................................................................... 14  
   do: .............................................................................. 17  
   select: aBlock ............................................................. 20  
   reject: aBlock ............................................................. 20  
   collect: aBlock .......................................................... 21  
   detect: aBlock ........................................................... 22  
   inject: thisValue into: binaryBlock ......................... 23  
Exercises ......................................................................... 27  

References  
Object-Oriented Design with Smalltalk — a Pure Object Language and its  
Environment, Ducasse, University of Bern, Lecture notes 2000/2001,  
http://www.iam.unibe.ch/~ducasse/WebPages/Smalltalk/ST00_01.pdf  

Squeak source code  

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

Array2D
  2 dimensional array

Heap
  A heap like you studied in data structures

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:
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>an 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>a Bag(1 1 2)</td>
</tr>
<tr>
<td>Set with: 1 with: 1 with: 2</td>
<td>a Set(1 2)</td>
</tr>
<tr>
<td>Bag new</td>
<td>a Bag()</td>
</tr>
<tr>
<td>OrderedCollection new</td>
<td>an 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

Squeak has special syntax to make it easy to create arrays. Other versions of Smalltalk do not have the dynamic array creation shown on the next slide. Some programmers avoid using this way creating arrays to reduce the effort needed to port their code other Smalltalks. This dynamic array creation is not part of the Smalltalk ANSI standard.

Literal Array Creation

Format:

#( element1  element2 … elementN )

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

Example

#( 1 2 'cat' ).

Literal Arrays and Variables

| array x |
x := 'cat'.
array := #( 1 'mouse' x ).
^array

Returns #(1 'mouse' #x)

As the above example shows variables including a variable in a literal array does not result in the value of the variable being storied in the array. A symbol for the variable name is stored.
Dynamic Array Creation

Format:
{ element1. element2. element3. element4. etc}

- Can contain variables
- The value of the variable at the time of creation is used

Examples

| array x |
array := { 1. 'mouse'. x }.
array

Returns #( 1 'mouse' 'cat' )

| array x |
array := { 1. 'mouse' . x }.
x := 'cat'.
array

Returns #( 1 'mouse' nil)
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>a SortedCollection($a $c $t)</td>
</tr>
<tr>
<td>#( 3 9 1 4 ) asSortedCollection</td>
<td>a SortedCollection(1 3 4 9)</td>
</tr>
<tr>
<td>#( 1 2 3 2 1) asBag</td>
<td>a Bag(1 1 2 2 3)</td>
</tr>
<tr>
<td>'hi mom' asBag</td>
<td>a Bag($m $m $o $h $i $ )</td>
</tr>
</tbody>
</table>

#( 3 9 1 4 ) asSortedCollection: [:x :y | x > y ]
Result
   a SortedCollection(9 4 3 1)

#( 3 9 1 4 ) asSortedCollection: [:x :y | x < y ]
Result
   a SortedCollection(1 3 4 9)

The block argument in asSortedCollection: is to return true if the first element should precede the second one
**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>collection size.</td>
<td>4</td>
</tr>
<tr>
<td>collection capacity</td>
<td>4</td>
</tr>
<tr>
<td>collection at: 2.</td>
<td>'b'</td>
</tr>
<tr>
<td>collection at: 1 put: 'cat'.</td>
<td></td>
</tr>
<tr>
<td>collection printString.</td>
<td>'#(&quot;cat&quot; &quot;b&quot; &quot;c&quot; &quot;d&quot;)'</td>
</tr>
<tr>
<td>collection := OrderedCollection new.</td>
<td></td>
</tr>
<tr>
<td>collection capacity.</td>
<td>10</td>
</tr>
<tr>
<td>collection size</td>
<td>0</td>
</tr>
</tbody>
</table>

The result shown below are the values returned by the statement on the same line as the result. You do not see the values unless you only execute the code up to that line.
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: $a.</td>
<td>an OrderedCollection($a )</td>
</tr>
<tr>
<td>Transcript show: a</td>
<td>an OrderedCollection($a &quot;cat&quot; 5)</td>
</tr>
<tr>
<td>a add: 'cat'.</td>
<td>an OrderedCollection($a &quot;cat&quot; 5 $d $o $g)</td>
</tr>
<tr>
<td>a add: 5.</td>
<td></td>
</tr>
<tr>
<td>Transcript show: a</td>
<td></td>
</tr>
<tr>
<td>a addAll: 'dog'.</td>
<td></td>
</tr>
<tr>
<td>Transcript show: a</td>
<td></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

removeAllFoundIn: aCollection
   Remove all elements in aCollection from the receiver
   Ignore all elements in aCollection not in the receiver
## Removing Examples

<table>
<thead>
<tr>
<th>Code</th>
<th>Output in Transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>data result original</td>
</tr>
<tr>
<td>data := original copy. data remove: 3. Transcript show: data; cr.</td>
<td>an OrderedCollection(4 2 1)</td>
</tr>
<tr>
<td>data := original copy. data remove: 5 ifAbsent: []. Transcript show:</td>
<td>an OrderedCollection(4 3 2 1)</td>
</tr>
<tr>
<td>data := original copy. data removeAll: #( 1 3). Transcript show:</td>
<td>an OrderedCollection(4 2)</td>
</tr>
<tr>
<td>data := original copy. data removeAllFoundIn: #( 1 6). Transcript</td>
<td>an OrderedCollection(4 3 2)</td>
</tr>
<tr>
<td>show: data; cr.</td>
<td></td>
</tr>
<tr>
<td>result := data remove: 4. Transcript show: result; cr.</td>
<td>result</td>
</tr>
</tbody>
</table>
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**

```plaintext
| 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:

```plaintext
| 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 with 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>do: aBlock</td>
<td>Evaluate aBlock with each of the receiver's elements as the argument.</td>
</tr>
<tr>
<td>select: aBlock</td>
<td>Evaluate aBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver, only those elements for which aBlock evaluates to true. Answer the new collection.</td>
</tr>
<tr>
<td>reject: aBlock</td>
<td>Evaluate aBlock with each of the receiver's elements as the argument. Collect into a new collection like the receiver only those elements for which aBlock evaluates to false. Answer the new collection.</td>
</tr>
<tr>
<td>collect: aBlock</td>
<td>Evaluate aBlock 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>detect: aBlock</td>
<td>Evaluate aBlock with each of the receiver's elements as the argument. Answer the first element for which aBlock evaluates to true. Signal an Error if none are found.</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>
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

iiaeae
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

| 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

| 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**

```ruby
| result |

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

^result

**Returned Value**

#(1 4 9 16 25)

```ruby
| 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]
```
More

There is a lot more to the collection classes in Squeak. Use the System browser to explore. We will come back to some of the classes, but will move on for now. Try the exercises on the next page.
Exercises

1. Show the order of evaluation of the subexpressions in the following:

   9 / 3 between: 10 - 2 squared and: 12.5 ceiling + 3

2. What happens when you select the word 'size' (without the quotes) in a workspace and type alt-m on a PC (command-m on Mac)?

3. What happens when you evaluate the following expressions?

   'cat' inspect
   'cat' class
   'cat' class inspect
   'cat' class browse
   'cat' class organization inspect

4. Find the method findTokens: and read its comment. From the comment explain what the following will return.

   'this,it a;test' findTokens: '; ,'

5. What class defines the method shuffled? What does it do?

6. What method in the string class tells you if one string contains another?

7. Write Smalltalk code to copy the 3rd to the 5th element in #( 2 4 1 3 5 ) to a new array.

8. Identify the messages and receivers in the following expression. What is the class of each receiver?

   (1 to: 20 by: 3) at: 4

9. What is the result of evaluation the expression:

   (1 to: 20 by: 3) asArray

10. Write Smalltalk code to take the vowels in 'this is an exercise' and place them in an OrderedCollection.

11. Write Smalltalk code to produce a list of the integers 1, 2, ..., 10 in random order.