CS 535 Object-Oriented Programming & Design
Fall Semester, 2003
Doc 8 Collections
Contents

References

VisualWorks Application Developer's Guide, doc/vwadg.pdf in the VisualWorks installation.

Reading

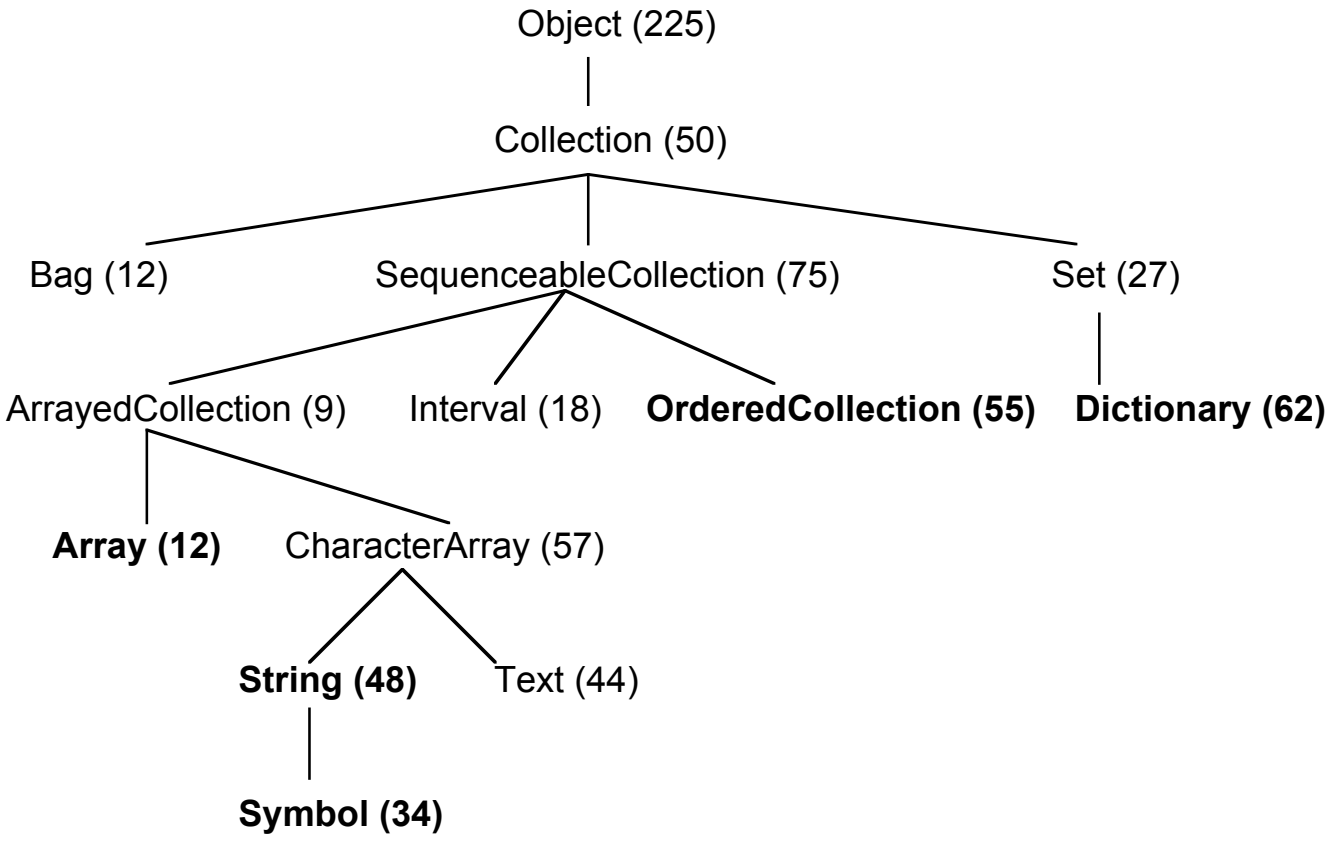
(DevGuide) Chapter 5 - Control Structures
Pages 110-112 Collection Iteration section

(DevGuide) Chapter 17 - Collections

(Beck) Chapter 5 - Collections pp 139-166

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.



Collection has 92 descendent classes

(N) indicates number of methods defined the class

Bold indicates commonly used 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

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.
```

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.

More Array Operations

```
numberOfElements := a size.

locationOfFiveInArray := a indexOf: 5.

jointList := a , b.          "Comma concatinates collections"

sublist := a
      copyFrom: 2
      to: 4.

copyList := a copyUpTo: 6.

copyWithout := jointList copyWithout:  2.

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

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

Much more common than arrays

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 Operations

```
b                    "Add elements to ordered collection, grow if needed"
    add: 2;
    add: 5.

secondElement := a at: 2.

firstElement := a first.

a
    at: 1
    put: 12.

jointList := a , b.

sublist := a
    copyFrom: 2
    to: 4.

copyList := a copyUpTo: 6.

copyWithout := jointList copyWithout:  2.

fiveIndex := a indexOf: 5.

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

a
    replaceAll: 3
    with: 12.

numberOfElements :=  a size.

a remove: 5
```

Size, Capacity & Growing

Size - number of elements in collection

Capacity - number of elements collection can hold without growing

```
| a |
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 Values

Recall CS 310

An item needs a hash value to be stored in a hash table

Object defines the method hash

Any object can be put into a dictionary

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 hash in a class you should redefine =

If you redefine = in a class is it recommended to redefine hash

Strings & Symbols

A String is an array of characters

'The cat in the hat'

Characters can be any Unicode character

Symbols are strings that are represented uniquely

Examples of symbols

#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"

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

Expression	Result printed
Array new: 5	#(nil nil nil nil nil)
OrderedCollection new: 5	OrderedCollection()
Array with: 2 with: 1	#(2 1)
Bag with: 1 with: 1 with: 2	Bag(1 1 2)
Set with: 1 with: 1 with: 2	Set(1 2)
Bag new	Bag()
OrderedCollection new	OrderedCollection()

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"

Converting

asArray
asBag
asSet
asOrderedCollection
asSortedCollection
asSortedCollection: aBlock
  Convert the receiver to the indicated collection

Examples

Expression	Result
'cat' asSortedCollection	SortedCollection ($a "16r0061" $c "16r0063" $t "16r0074")
#(3 9 1 4) asSortedCollection	SortedCollection(1 3 4 9)
#(1 2 3 2 1) asBag	Bag(1 1 2 2 3)
'hi mom' asBag	Bag (Core.Character space $o "16r006F" $h "16r0068" $i "16r0069" $m "16r006D" $m "16r006D")

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

Expression	Result
#(3 9 1 4) asSortedCollection: [:x :y | x > y]	SortedCollection(9 4 3 1)
#(3 9 1 4) asSortedCollection: [:x :y | x < y]	SortedCollection(1 3 4 9)
#(3 9 1 4) asSortedCollection	SortedCollection(1 3 4 9)
#('dog' 'mat' 'bee') asSortedCollection	SortedCollection('bee' 'dog' 'mat')
#($2 $a $A $w) asSortedCollection	SortedCollection ($2 $A $a $w)
'cathat' asSortedCollection	SortedCollection($a $a $c $h $t $t)

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

```
#( 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

	collection		Result
collection := #('a' 'b' 'c' 'd').			
Transcript print: collection size.			4
Transcript print: collection capacity			4
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

a		Result on the transcript
a := OrderedCollection with: $a.		
Transcript show: a		OrderedCollection($a)
a add: 'cat'.		
a add: 5.		
Transcript show: a.		OrderedCollection($a "cat" 5)
a addAll: 'dog'.		
Transcript show: a		OrderedCollection($a "cat" 5 $d $o $g)

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

| data result original |	Output in Transcript
original := #(4 3 2 1) asOrderedCollection.	
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
occurencesOf: anElement

Examples

Expression	Result
#(1 6) isEmpty	false
'cat' includes: $o	false
'mom' occurrencesOf: $m	2
#(1 3 2 4 3) occurrencesOf: 3	2

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

do: aBlock	Evaluate aBlock with each of the receiver's elements as the argument.
select: aBlock	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.
reject: aBlock	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.
collect: aBlock	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.
detect: aBlock	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.
inject: initialValue into: binaryBlock	Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value of the argument, thisValue, and the receiver as block arguments.

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

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

Digit	Frequency
0	30
1	15
2	19
3	10
4	10
5	14
6	19
7	7
8	14
9	20

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

```
| 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

Partial	Number
0	1
1	2
3	3
6	4
10	5

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]
```

Some Useful Enumerations on Sequenceable Collections

with:do:

enumeration over two collections

Collections must be of the same size!

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

Returns
OrderedCollection (3 7 11 15 19)

do:seperatedBy:

Performs a seperatedBy: block between performing the do: block.

```
#(2 4 6 8)
    do: [:each | Transcript print: each ]
    separatedBy: [Transcript show: ', '].
Transcript
    cr;
    flush
```

Prints on Transcript:

2, 4, 6, 8

fold:

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.

```
#(1 2 3) fold: [:a : b | a + b]
```

returns
6

```
#( 'A' 'cat' 'in' 'the' 'hat' ) fold: [:a :b | a , ' ' , b]
```

returns

'A cat in the hat'

piecesCutWhere:do:

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

do: block is performed on each piece

```
'A sentence. Another sentence... Yet another sentence.'
        piecesCutWhere:
            [:each :next |
            each = $. and: [next = Character space]]
        do:
            [:each |
            Transcript
                show: each printString;
                cr]
```

Prints

'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]
```

Prints

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

runsFailing:do:

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

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

Prints

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

Blocks and Returns

When a block evaluates a return (^) it exits the method the block was defined in

Example

```
Smalltalk.CS535 defineClass: #BlockExample
	superclass: #{Core.Object}
	etc

doExample
	Transcript
		show: 'Start doExample';
		cr.
	self start.
	Transcript
		show: 'End doExample';
		cr.!

evaluate: aBlock
	Transcript
		show: 'Start evaluate';
		cr.
	aBlock value.
	Transcript
		show: 'End evaluate';
		cr.!

start
	Transcript
		show: 'Start start';
		cr.
	self evaluate: [^nil].
	Transcript
		show: 'End start';
		cr.! !
```

Running the Example

Evaluate:

BlockExample new doExample

The output in the Transcript is:

Start doExample
Start start
Start evaluate
End doExample