

Collections

Collection ()

Bag ('contents')

SequenceableCollection ()

ArrayedCollection ()

LinkedList ('firstLink' 'lastLink')

Semaphore ('excessSignals')

Interval ('start' 'stop' 'end')

OrderedCollection ('firstIndex' 'lastIndex')

SortedCollection ('sortBlock')

Set ('tally')

Dictionary ()

IdentitySet ()

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

161

Abstract Class as Template

Most operations defined in terms of
do:

- select:, collect:, inject:into:,
detect:ifAbsent:, size

A program skeleton

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

162

Abstract class as type

All collections understand same protocol

do:	iterate
select:	subcollection
collect:	transformed collection
inject:into:	reduce collection to value

Nonuniform Collection Protocol

- add: and remove: are defined by collections whose size can change--- Set, OrderedCollection, Bag, etc.
- ArrayedCollection and Interval do not implement add: and remove:
- at: and at:put: are defined by SequenceableCollection and Dictionary, but not by Set or Bag.

Collection Protocols

Collection do:, size, select:, collect:,
inject:into:, includes:

- *ChangeableCollection* add:
addAll: remove: removeAll:
- *SequenciableCollection* at: at:put
first last , copyFrom:to:

Collection Menagerie

Array, String

Seq.

Symbol

Seq., read-only, unique

OrderedCollection

Seq., Changeable., a sequence

Set, Bag Changeable

Collection Menagerie

Dictionary

at: and at:put: with any object as key

Interval

Seq., numeric, read-only

RunArray

Seq., compact encoding

SequenceableCollection Example

x := collection first.

collection do: [:each | x := x min:
each]

x := collection inject: collection first
into: [:result :each | result min:
each]

SequenceableCollection

= aCollection

- self size = aCollection size ifFalse: [^false].
- 1 to: self size do: [:i | (self at: i) = (aCollection at: i) ifFalse: [^false].
- ^true

Dictionary Example

employees

at: 'John Doe' put: 319426621;

at: 'Jane Smith' put: 321654321

employees keys

employees values

(continued)

employees do: [:ssNum | ...]

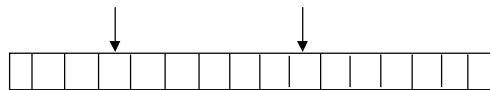
employees keysAndValuesDo: [:key
:value |

- Transcript show: key; show: ' has number ';
- show: (value printString); cr]

OrderedCollection

Instance variables:

firstIndex, lastIndex



A changeable sequenceable collection.

Supports add:, remove:, at:, at:put:

OrderedCollection

at: anInteger

anInteger isInteger ifFalse: [^...].

(anInteger < 1 or: [anInteger + firstIndex - 1
> lastIndex])

ifTrue: [^...]

^ super at: anInteger + firstIndex - 1

OC::do:

do: aBlock

firstIndex to: lastIndex do:

[:index | aBlock value: (self basicAt:
index)].

size

^ lastIndex - firstIndex + 1

OC::add:

add: newObject

"Include newObject as one of the receiver's elements. Answer newObject."

^ self addLast: newObject

OC::add:

addLast: newObject

"Add newObject to the end of the receiver. Answer newObject."

lastIndex = self basicSize ifTrue: [self makeRoomAtLast].

lastIndex := lastIndex + 1.

self basicAt: lastIndex put: newObject.

^ newObject

Using OrderedCollections

Suppose that list is an ordered collection.
We want newList to be the reverse of list.

solution 1

```
newList := OrderedCollection new.  
list do: [:each | newList addFirst: each]
```

(continued)

solution 2

```
newList := OrderedCollection new: list  
size.  
index := list size.  
list do: [:each | newList at: index  
put: each.  
index := index - 1]
```

Interval

Instance variables: start, stop, step

1 to: 1000

1 to: 1000 by: 3

1 to: 1000 do: [:each | sum := sum + each]

Interval

Number

to: stop

^ Interval from: self to: stop by: 1

Interval do:

do: aBlock

| n end |

n := 0.

end := self size - 1.

[n <= end]

whileTrue:

[aBlock value: start + (step * n).

n := n + 1]

Interval at: and at:put:

at: anInteger

(anInteger >= 1 and: [anInteger <= self
size])

ifTrue: [^start + (step * (anInteger - 1))]

ifFalse: [^self subscriptBoundsError:
anInteger]

Interval at:put:

at: anInteger put: anObject

"Provide an error notification that storing into an Interval is not allowed. "

self error: 'you can not store into an interval'

Set

		3		19	\$x			11	'hi!'
--	--	---	--	----	-----	--	--	----	-------

A hash table. nil means an unused entry.

Instance variables: tally

Hash

- Sets and Dictionaries depend on the **hash** of their elements.
- Each object has an integer hash value.
- Some classes redefine hash.
- Invariant: Equal (=) objects hash equally.

Classic Set Bugs

Some classes redefine hash.

Invariant: Equal (=) objects hash equally.

Bug 1: redefine = but not hash for some class.

- -- aSet add: x.
- x = y
- ifTrue: [(aSet includes: y)
- ifFalse: [self error]]

Set bugs

Bug 2: Define = so it depends on state of the object.

- Put object in set, then change its state, then try to find it!

Rule of thumb: only redefine = if object is immutable.

Set do:

do: aBlock

tally == 0 ifTrue: [^self].

1 to: self basicSize do:

[:index |

| elem |

(elem := self basicAt: index) == nil

ifFalse: [aBlock value: elem]]

Set includes:

includes: anObject

"Answer whether anObject is one of the receiver's elements."

```
^(self basicAt: (self findElementOrNil:  
anObject)) notNil
```

Set

findElementOrNil: anObject

"Answer the index of the argument anObject, if present, or the index of a nil entry where anObject would be placed."

```
| index length probe pass |
```

```
length := self basicSize.
```

```
pass := 1.
```

(continued)

```
index := self initialIndexFor: anObject
hash boundedBy: length.
[(probe := self basicAt: index) == nil or:
 [probe = anObject]]
  whileFalse: [...].
^index
```

(continued)

```
whileFalse: [
  (index := index + 1) > length
  ifTrue:
    [index := 1.
     pass := pass + 1.
     pass > 2 ifTrue: [^self grow
      findElementOrNil: anObject]]]
```

Classic Collection Bugs

add: returns argument

- (aSet add: 3) add: 2 --- wrong
- aSet add: 3; add: 2 --- right

How are these different?

- Array with: 3 with: 17 with: -9
- #(3 17 -9)

Keeping Track of Time

Bug:

- 1. Forget to enter timecard
- 2. Raise salary
- 3. Notice that timecard is missing and enter it.

Keeping Track of Time

Problem: how do you ensure that each transaction is processed with rules in effect at the time it took place?

Keeping Track of Time

Things that are hard with current payroll design:

- • Make a graph of salary paid per month for each person.
- • Make a graph of vacation time taken per month for the entire company.

ValueWithHistory

I represent a (probably numerical) value that changes over time. I can answer my value at any point in time (usually a date). I can update my value at any point in time, and also add a number to my value from any point in time on into the future. My value does not change continuously, but changes at discrete points in time.

ValueWithHistory

Instead of storing a value in a variable,
store it in a ValueWithHistory.
earnings := ValueWithHistory zero.
earnings at: today add: 50.
earnings starting: endOfYear become: 0.

ValueWithHistory protocol

at: aDate - return value at aDate
at: aDate add: anAmount - add anAmount
to value at aDate and
at all times in the future
starting: aDate become: anAmount - set
value from aDate to the next
specified time to anAmount

ValueWithHistory variables

date <SortedCollection of: Date>
value <OrderedCollection of: Number>
The i'th element of value matches the i'th
element of date. The date collection
indicates when the value takes on the
next element of the value collection.

ValueWithHistory

initialize

```
date := SortedSequence new.  
value := OrderedCollection new.
```

ValueWithHistory

at: aDate

"Return value at the date"

| index |

index := date

indexOfStartOfIntervalContaining: aDate.

index = 0 ifTrue: [self error: 'date is too early'].

^value at: index

SortedSequence

SortedSequence is a subclass of
SortedCollection with one method:

**indexOfStartOfIntervalContaining:
anElement**

"Return the index of anElement or, if it is
not present, of the index of the largest
element smaller than it."

SortedSequence

**indexOfStartOfIntervalContaining:
anElement**

self isEmpty ifTrue: [^0].

^(self indexForInserting: anElement) -
firstIndex

ValueWithHistory

at: aDate add: anAmount | start |
start := (self indexForAccessing: aDate).
start
to: value size
do: [:each | value at: each put: (value at:
each) + anAmount]

ValueWithHistory

indexForAccessing: aDate
"Return index of slot for aDate, creating it if
necessary."
| index | index := date
indexOfStartOfIntervalContaining: aDate.
index = 0 ifTrue: [date add: aDate. value
addFirst: 0. ^1].
(date at: index) = aDate ifTrue: [^index].

(continued)

date add: aDate.
value add: (value at: index)
beforeIndex: index + 1.
^index + 1

Collection methods

add: anElement
adds to end of OrderedCollection, to
position in sort-order of
SortedCollection, to random position
in Set

OrderedCollection methods

addFirst: anElement

adds to beginning of OrderedCollection

add: anElement beforeIndex: anInteger

insert element at location, moving what is there

ValueWithHistory

starting: aDate become: aValue

"Change value so that it becomes
aValue at aDate."

| index |

index := self indexForAccessing: aDate.

value at: index put: aValue

Moral

It is often best not to make instance variable be simple object like number or string.

- 1) Make it be a domain object.
(Money instead of Number)
- 2) Make it be a holder.
(ValueWithHistory instead of Number or Money)