

Ю.А. Кирютенко, В.А. Савельев

Объектно-ориентированное программирование
и язык Smalltalk.

Класс Collection и его подклассы.

Часть 2

Ростов-на-Дону

1997

Объектно-ориентированное программирование
и язык Smalltalk.

Класс Collection и его подклассы.

Часть 2

Аннотация

Методическая разработка посвящена современному направлению в программировании — объектно-ориентированной методологии программирования и языку Smalltalk и является продолжением вышедших ранее двух методических разработок по объектно-ориентированному программированию и языку Smalltalk, посвященных общим концепциям и синтаксису языка, интерфейсу пользователя и среде программирования. В двух частях этой методической разработки описывается протокол и реализация системного класса Collection и важнейших из его многочисленных подклассов. Всюду далее при описании классов языка Smalltalk мы будем иметь ввиду их реализацию в языке Smalltalk/V фирмы Digitalk (США).

Методическая разработка предназначена для студентов 3–5 курсов механико-математического факультета и слушателей ФПК.

Печатается в соответствии с решением кафедры математического анализа Ростовского государственного университета, протокол № 10 от “5” июня 1996 года.

Настоящие методические указания набраны в системе L^AT_EX с использованием кириллических шрифтов семейства LH (дизайнеры О. Лапко и А. Ходулев).

1 Класс IndexedCollection

Начнем изучение следующего подкласса класса `Collection` — класса `IndexedCollection`, который является абстрактным классом, представляющим наборы, элементы которых упорядочены и могут именоваться извне с помощью целых индексов. Поэтому, говоря здесь об экземплярах этого класса, всегда имеются в виду экземпляры его подклассов, которые полностью поддерживают определенный здесь общий протокол. Класс `IndexedCollection` является подклассом класса `Collection` и поддерживает протокол доступа, копирования, перечисления элементов из набора, когда известно, что существует порядок среди его элементов. Все экземпляры класса `IndexedCollection` наследуют из класса `Object` сообщения для работы с индексированными элементами такие, как `at:`, `at:put:`, `size`. Дополнительно, класс `IndexedCollection` поддерживает размещение заданного объекта во всех задаваемых позициях (`atAll:put:`), размещение заданного объекта во всех позициях набора (`atAllPut:`), замену элементов из одного набора на элементы из другого набора (`replaceFrom:to:with:`, `replaceFrom:to:with:startingAt:`).

IndexedCollection instance protocol

`atAll: aCollection put: anObject` — заменяет каждый элемент набора-приемника с ключом из аргумента `aCollection` (индексом или другим внешним ключом) на второй аргумент `anObject`.

`atAllPut: anObject` — заменяет каждый элемент приемника на аргумент `anObject`.

`first` — возвращает первый элемент приемника, сообщает об ошибке, если приемник пуст. `item[last]` — возвращает последний элемент приемника, сообщает об ошибке, если приемник пуст.

`indexOf: anElement` — возвращает первый индекс элемента из набора-приемника равного аргументу `anElement`, если такого элемента нет, возвращает 0.

`indexOf: anElement ifAbsent: aBlock` — возвращает индекс первого элемента из набора-приемника равного аргументу `anElement`, если такого элемента нет, возвращает результат вычисления блока `aBlock`.

`replaceFrom: start to: stop with: replacementCollection` — возвращает набор-приемник, в котором, начиная с индекса `start` и заканчивая индексом `stop`, элементы заменены на элементы набора-аргумента `replacementCollection`; в `replacementCollection` должно быть точно `stop - start + 1` элементов.

`replaceFrom: start to: stop`

`with: replacementCollection startingAt: repStart`

— возвращает набор-приемник, в котором, начиная с индекса `start` и заканчивая индексом `stop`, элементы заменены на элементы набора-аргумента `replacementCollection`, начиная с индекса `repStart`.

Рассмотрим примеры реализации этих сообщений, применительно к экземплярам класса `String`, который является подклассом класса `IndexedCollection`.

выражение	результат
<code>'aaaaaaaa' size</code>	8
<code>'aaaaaaaa' at: (2 to: 8 by: 2) put: \$b</code>	<code>'abababab'</code>
<code>'abababab' indexOf: \$b</code>	2
<code>'aaaaaaaa' atAllPut: \$b</code>	<code>'bbbbbbbb'</code>
<code>'This string' first</code>	<code>\$T</code>
<code>'This string' last</code>	<code>\$g</code>
<code>'ABCDEFGH' indexOf: \$F</code>	6
<code>'ABCDEFGH' indexOf: \$M ifAbsent: [0]</code>	0
<code>'ABCDEFGH' indexOf: \$C ifAbsent: [0]</code>	3
<code>'The cow eat' replaceFrom: 5 to: 7 with: 'dog'</code>	<code>'The dog eat'</code>

Все эти примеры аналогично создаются и для экземпляров других подклассов класса `IndexedCollection`. Еще одно общее замечание: возможно при таких преобразованиях следует сохранять набор-оригинал, делая предварительно его копию. Протокол копирования класса `IndexedCollection` поддерживает копирование последовательности элементов в набор, копирование набора с заменой его части, копирование набора с удалением части элементов, копирование набора с присоединением новых элементов.

, **anIndexedCollection** — конкатенация (объединение) наборов; возвращает набор-приемник, в который по порядку добавлены все элементы аргумента **anIndexedCollection**.

copyFrom: start to: stop — возвращает копию части приемника сообщения, состоящую из его элементов с индексами от **start** до **stop**.

copyReplaceFrom: start to: stop with: replCollection — когда числа **start** и **stop** взяты в допустимых пределах, возвращает копию приемника сообщения, которая состоит из конкатенации части приемника сообщения от его начала до элемента с индексом **start** – 1, набора **replCollection** и части приемника сообщения от элемента с индексом **stop** + 1 до конца.

copyWith: newElement — возвращает копию приемника сообщения, в конец которого добавлен еще один элемент **newElement**.

copyWithout: oldElement — возвращает копию приемника сообщения, из которого удален первый элемент равный **oldElement**.

Последние два сообщения из протокола копирования полезны при копировании экземпляров класса **Array** с добавлением или удалением элементов. Например,

выражение	результат
<code> #(one two three) copyWith: #four</code>	<code> (one two three four)</code>
<code> #(one two three) copyWithout: #two</code>	<code> (one three)</code>

Так как элементы из экземпляров класса **IndexedCollection** упорядочены, то возможно их перечисление — вызов по порядку, начиная с первого и до последнего. Возможно перечисление и в обратном порядке (от последнего к первому), для чего используется сообщение **reverseDo: aBlock**. Возможно так же совместное перечисление двух экземпляров класса **IndexedCollection**, при котором пара элементов, по одному из каждого набора, используется в качестве аргументов вычисляемого блока.

findFirst: aBlock — вычисляет одно аргументный блок **aBlock** с каждым элементом из приемника сообщения в качестве аргумента, начиная с первого; возвращает индекс первого элемента, для которого вычисление блока завершается возвращением значения **true**.

findLast: aBlock — вычисляет одно аргументный блок **aBlock** с каждым элементом из приемника сообщения в качестве аргумента, начиная с первого; возвращает индекс последнего элемента, для которого вычисление блока завершается возвращением значения **true**.

reverseDo: aBlock — вычисляет одно аргументный блок **aBlock** с каждым элементом из приемника сообщения в качестве аргумента, начиная с последнего (все происходит в обратном порядке по сравнению с выполнением сообщения **do: aBlock**).

with: aIndexedCollection do: aBlock — вычисляет двух аргументный блок **aBlock** для каждого элемента из приемника сообщения и соответствующего элемента из набора-аргумента **aIndexedCollection** в качестве аргументов блока; размеры приемника сообщения и набора-аргумента должны быть равны.

Следующее выражение создает тот же самый словарь **opposites**, который ранее уже использовался как пример.

```
opposites := Dictionary new.  
#(come cold front hot push stop)  
  with: #(go hot back cold pull start)  
  do: [:key :value | opposites at: key put: value].
```

Словарь **opposites** имеет сейчас 6 элементов, каждый из которых — экземпляр класса **Association**.

Как мы уже знаем, любой экземпляр класса **IndexedCollection** можно преобразовать в экземпляр класса **Array**, если послать ему сообщение **asArray**, наследуемое из класса **Collection**. После приведенных примеров и разъяснений, в реализации методов этого класса читатель, как мы надеемся, сможет разобраться самостоятельно, используя браузер иерархии классов.

2 Класс OrderedCollection

Элементы экземпляра класса `OrderedCollection` упорядочены последовательностью, в которой они добавляются и удаляются и хранят упорядоченную (неограниченную) последовательность элементов. При этом экземпляры класса `OrderedCollections` расширяемы, и выигрыш в эффективности достигается выделением дополнительного пространства для элементов последовательности, когда для их размещения не хватает места. Две переменные экземпляра, `startPosition` и `endPosition`, указывают на первый и последний элементы в экземпляре. Элементы доступны по внешним ключам-индексам. Протоколы доступа, добавления и удаления дополняются ссылками на первый и последний, на предыдущий и последующий элементы.

Экземпляр класса `OrderedCollection` может выступать как стек или очередь. Стек — это последовательный список, в котором удаление и добавление элементов производятся на одном и том же конце списка, называемом вершиной стека. Часто о стеке говорят, что это список с порядком «последним пришел — первым ушел» (*'last-in first-out'* или, сокращенно, *lifo*-список).

Обычные выражения	Сообщения класса <code>OrderedCollection</code>
<code>push newObject</code>	<code>addLast: newObject</code>
<code>pop</code>	<code>removeLast</code>
<code>top</code>	<code>last</code>
<code>empty</code>	<code>isEmpty</code>

Очередь — это последовательный список, в котором удаление элементов производятся в конце списка, а добавление элементов производятся в начале списка. Часто об очереди говорят, что это список с порядком «первым пришел — первым ушел» (*'first-in first-out'* или, сокращенно, *fifo*-список).

обычные выражения	сообщения класса <code>OrderedCollection</code>
<code>add newObject</code>	<code>addLast: newObject</code>
<code>delete</code>	<code>removeFirst</code>
<code>front</code>	<code>first</code>
<code>empty</code>	<code>isEmpty</code>

2.1 Описание протокола

Сообщение **add**: в классе **OrderedCollection** добавляет в набор аргумент сообщения как последний элемент; сообщение **remove.ifAbsent**: удаляет аргумент сообщения из набора, если такового нет — возвращает результат выполнения блока. В дополнение к сообщениям, наследуемым из своих суперклассов, этот класс определяет такие сообщения

OrderedCollection instance protocol

after: oldObject — возвращает элемент из приемника сообщения, стоящий после элемента **oldObject**, если нет элемента **oldObject** или после него нет элемента, сообщает об ошибке.

before: oldObject — возвращает элемент из приемника сообщения, стоящий до элемента **oldObject**, если нет элемента **oldObject** или до него нет элемента, сообщает об ошибке.

add: newObject after: oldObject — добавляет в приемник элемент **newObject** после элемента **oldObject**, возвращает **newObject**; если элемента **oldObject** в приемнике сообщения нет, сообщает об ошибке.

add: newObject before: oldObject — добавляет в приемник элемент **newObject** непосредственно перед элементом **oldObject**, возвращает **newObject**; если элемента **oldObject** в приемнике сообщения нет, сообщает об ошибке.

addAllFirst: anOrderedCollection — добавляет каждый элемент из набора-аргумента **anOrderedCollection** в начало приемника сообщения, возвращает **anOrderedCollection**.

addAllLast: anOrderedCollection — добавляет каждый элемент из набора-аргумента **anOrderedCollection** в конец приемника сообщения, возвращает **anOrderedCollection**.

addFirst: newObject — добавляет элемент **newObject** в начало приемника сообщения, возвращает **newObject**.

addLast: newObject — добавляет элемент **newObject** в конец приемника сообщения, возвращает **newObject**.

removeFirst — удаляет первый элемент в приемнике сообщения и возвращает его; если приемник пуст, сообщает об ошибке.

`removeLast` — удаляет последний элемент из приемника сообщения и возвращает его; если приемник пуст, сообщает об ошибке.

`removeIndex: anIndex` — удаляет элемент из приемника в позиции `anIndex`, возвращает приемник; если `anIndex` вне возможных границ - сообщает об ошибке.

2.2 Реализация класса

Приведем почти полную реализацию класса `OrderedCollection`, исключив из нее лишь некоторые частные методы.

```
IndexedCollection subclass: #OrderedCollection
  instanceVariableNames:
    'startPosition endPosition contents '
  classVariableNames: ''
  poolDictionaries: ''
```

`OrderedCollection` class methods

```
new
  ^self new: 12

new: anInteger
  ^(super new) initPositions: anInteger
```

`OrderedCollection` methods

```
, aCollection
  ^self copy
    addAll: aCollection;
    yourself

add: anObject
  endPosition = contents size
    ifTrue: [self putSpaceAtEnd].
  endPosition := endPosition + 1.
  contents at: endPosition put: anObject.
```

^ anObject

add: newObject after: oldObject

| index |

index := 1.

[index <= self size]

whileTrue: [

oldObject = (self at: index)

ifTrue: [^ self add: newObject afterIndex: index].

index := index + 1].

^ self errorAbsentElement

add: anObject afterIndex: anInteger

self putSpaceAfter: anInteger.

^ self at: anInteger + 1 put: anObject

add: newObject before: oldObject

| index |

index := 1.

[index <= self size]

whileTrue: [

oldObject = (self at: index)

ifTrue: [^ self add: newObject beforeIndex: index].

index := index + 1].

^ self errorAbsentElement

add: anObject beforeIndex: anInteger

self putSpaceAfter: anInteger - 1.

^ self at: anInteger put: anObject

addAllFirst: aCollection

| index |

index := aCollection size.

[index <= 0]

whileFalse: [

self addFirst: (aCollection at: index).

index := index - 1].

^ aCollection

addAllLast: aCollection

```

| index size |
size := aCollection size.
index := 1.
[index <= size]
  whileTrue: [
    self addLast: (aCollection at: index).
    index := index + 1].
^ aCollection

```

```

addFirst: anObject
  startPosition = 1
  ifTrue: [self putSpaceAtStart].
  startPosition := startPosition - 1.
  contents at: startPosition put: anObject.
^ anObject

```

```

addLast: anObject
  endPosition = contents size
  ifTrue: [self putSpaceAtEnd].
  endPosition := endPosition + 1.
  contents at: endPosition put: anObject.
^ anObject

```

```

after: anObject
  ^self
  after: anObject
  ifNone: [^self errorAbsentElement]

```

```

after: anObject ifNone: aBlock
  | index |
  index := startPosition.
  [index < endPosition]
    whileTrue: [
      anObject = (contents at: index)
      ifTrue: [^ contents at: index + 1].
      index := index + 1].
  ^ aBlock value

```

```

at: anInteger
  | index |

```

```
index := anInteger + startPosition - 1.  
(startPosition <= index and: [index <= endPosition])  
  ifFalse: [  
    ^self errorInBounds: anInteger].  
^contents at: index
```

```
at: anInteger put: anObject  
| index |  
index := anInteger + startPosition - 1.  
(startPosition <= index and: [index <= endPosition])  
  ifFalse: [  
    ^self errorInBounds: anInteger].  
contents at: index put: anObject.  
^anObject
```

```
before: anObject  
  ^self before: anObject  
    ifNone: [^self errorAbsentElement]
```

```
before: anObject ifNone: aBlock  
| index |  
index := startPosition + 1.  
[index <= endPosition]  
  whileTrue: [  
    anObject == (contents at: index)  
    ifTrue: [^contents at: index - 1].  
    index := index + 1].  
^aBlock value
```

```
copyFrom: beginning to: end  
| answer |  
(answer := self species new: self size)  
  startPosition: 1  
  endPosition: end - beginning + 1.  
^answer  
  replaceFrom: 1  
  to: end - beginning + 1  
  with: self  
  startingAt: beginning
```

Сообщение перечисления с селектором `do:` переопределяется для того, чтобы обеспечивать большую эффективность вычислений, чем у метода из класса `IndexedCollection`.

```
do: aBlock  
  | index |  
  index := startPosition - 1.  
  [(index := index + 1) <= endPosition]  
    whileTrue: [aBlock value: (contents at: index)]
```

```
includes: anObject  
  | index |  
  index := startPosition - 1.  
  [(index := index + 1) > endPosition]  
    whileFalse: [  
      anObject = (contents at: index)  
      ifTrue: [^ true].  
    ]  
  ^ false
```

```
remove: anObject ifAbsent: aBlock  
  | index |  
  index := startPosition.  
  [index <= endPosition]  
    whileTrue: [  
      anObject = (contents at: index)  
      ifTrue: [  
        self removeIndex: index.  
        ^ anObject].  
      index := index + 1].  
  ^ aBlock value
```

```
removeFirst  
  | answer |  
  startPosition > endPosition  
    ifTrue: [^ self errorAbsentElement].  
  answer := contents at: startPosition.  
  contents at: startPosition put: nil.  
  startPosition := startPosition + 1.  
  ^ answer
```

```

removeIndex: anInteger
| index |
(anInteger between: startPosition and: endPosition)
  ifFalse: [^self errorAbsentElement].
index := anInteger.
[index < endPosition]
  whileTrue: [
    contents at: index
      put: (contents at: index + 1).
    index := index + 1].
contents at: endPosition put: nil.
endPosition := endPosition - 1

```

```

removeLast
| answer |
startPosition > endPosition
  ifTrue: [^self errorAbsentElement].
answer := contents at: endPosition.
contents at: endPosition put: nil.
endPosition := endPosition - 1.
^answer

```

```

replaceFrom: start to: stop with: aCollection
| finalSize size index |
size := aCollection size.
finalSize := self size + size - (stop - start + 1).
finalSize > contents size
  ifTrue: [self growTo: finalSize + (finalSize // 3 + 10)].
self
  startPosition: startPosition
  endPosition: contents size.
self
  replaceFrom: start + size
  to: finalSize
  with: self
  startingAt: stop + 1.
self
  replaceFrom: start
  to: start + size - 1

```

```

    with: aCollection
    startingAt: 1.
self
    startPosition: startPosition
    endPosition: startPosition + finalSize - 1.
index := endPosition + 1.
[index <= contents basicSize]
    whileTrue: [
        contents at: index put: nil.
        index := index + 1].
^self

size
    ^endPosition - (startPosition - 1)

```

3 Класс SortedCollection

Класс `SortedCollection` — подкласс класса `OrderedCollection`. Элементы в экземпляре класса `SortedCollection` упорядочиваются с помощью функции двух переменных. Эта функция представляется двух аргументным блоком, который называется блоком сортировки. В такие наборы новый элемент можно добавить только с помощью сообщения `add:`. Определение места вставки нового элемента вычисляется с помощью метода «пузырьковой сортировки», хорошо известного в программировании. Сообщения `addFirst:`, `addLast`, `at:put:`, посредством которых программист может определить место нового элемента в наборе, не могут посылаться экземплярам класса `SortedCollection`.

Экземпляр класса `SortedCollection` может быть создан посылкой классу `SortedCollection` сообщения `sortBlock: aBlock`, в котором аргумент сообщения `aBlock` — это двух аргументный блок, устанавливающий будущий порядок элементов в создаваемом наборе. Например,

```
SortedCollection sortBlock: [:a :b | a <= b].
```

Приведенный в этом примере блок сортировки создается по умолчанию для экземпляра класса `SortedCollection`, когда новый экземпляр создается посредством посылки классу сообщения `new`. Следующие примеры демонстрируют 4-е способа образования нового экземпляра

класса `SortedCollection`, из которых последние два наследуются их класса `Collection`.

`SortedCollection new`.

`SortedCollection sortBlock: [:a :b | a > b]`.

`anyCollection asSortedCollection`.

`anyCollection asSortedCollection: [:a :b | a > b]`.

Есть возможность изменять определенный ранее блок сортировки, используя сообщение к экземплярам класса `SortedCollection`; при этом, когда меняется блок сортировки, элементы набора переупорядочиваются в соответствии с новой функцией сортировки.

`SortedCollection` instance protocol

`sortBlock` — возвращает блок сортировки приемника сообщения.

`sortBlock: aBlock` — меняет на аргумент `aBlock` блок сортировки в наборе-приемнике и пересортировывает приемник.

Обратим внимание на то, что одно и то же сообщение (`sortBlock: aBlock`) посылается классу `SortedCollection`, когда необходимо создать новый экземпляр с указанным блоком сортировки (смотри примеры, приведенные выше), и экземпляру класса `SortedCollection`, когда необходимо изменить критерий сортировки элементов данного экземпляра.

Рассмотрим пример. Предположим, что мы желаем создать алфавитно-упорядоченный список имен детей некоторого класса.

`ch := SortedCollection new`.

По умолчанию у набора `ch` образовался блок сортировки

`[:a :b | a <= b]`

Элементами этого набора могут быть экземпляры классов `String` и `Symbol`, которые, как будет показано позже, отвечают на сообщения `<`, `>`, `<=`, `>=`.

Выражение	Результат
ch add: #Joe	Joe
ch add: #Bill	Bill
ch add: #Alice	Alice
ch	SortedCollection(Alice Bill Joe)
ch sortBlock: [:a :b a > b]	SortedCollection(Joe Bill Alice)
ch add: #Dan	Dan
ch	SortedCollection(Joe Dan Bill Alice)

3.1 Реализация класса

Реализован класс `SortedCollection` таким образом, что сообщения `add:after:`, `add:before:`, `addAllFirst:`, `addAllLast:`, `addFirst:`, `addLast:`, `at:put:`, наследуемые из суперкласса `OrderedCollection` описываются посредством `^ self invalidMessage` (и мы их в нижеследующей реализации не показываем, как и частные методы).

```
OrderedCollection subclass: #SortedCollection
  instanceVariableNames:
    'sortBlock '
  classVariableNames: ''
  poolDictionaries: ''
```

`SortedCollection` class methods

```
new: anInteger
  ^ (super new: anInteger) sortBlock: [:a :b | a <= b]

sortBlock: aBlock
  ^ (super new: 10) sortBlock: aBlock
```

`SortedCollection` methods

```
add: anObject
  | index element |
  endPosition = contents size
  ifTrue: [self putSpaceAtEnd].
  index := endPosition.
  endPosition := endPosition + 1.
```

```

[index < startPosition]
whileFalse: [
    element := contents at: index.
    (sortBlock
        value: anObject
        value: element)
    ifFalse: [
        ^ contents at: index + 1
        put: anObject].
    contents at: index + 1 put: element.
    index := index - 1].
^ contents at: index + 1 put: anObject

```

```

addAll: aCollection
aCollection do: [ :element |
    super addLast: element].
self reSort.
^ aCollection

```

```

copyFrom: beginning to: end
| answer |
answer := self species new: self size.
answer sortBlock: sortBlock.
beginning to: end do: [:i |
    answer add: (self at: i)].
^ answer

```

```

sortBlock
^ sortBlock

```

```

sortBlock: aBlock
sortBlock := aBlock.
self reSort "происходит пересортировка набора"

```

4 Класс FixedSizeCollection

Класс `FixedSizeCollection` — абстрактный подкласс класса `Collection`, экземпляр подклассов которого — это набор элементов с внешними

ключами-индексами фиксированного, определяемого при образовании размера. Среди подклассов класса `FixedSizeCollection` в системе `Smalltalk` отметим подклассы `Interval`, `Array`, `String` и его подкласс `Symbol`.

```
IndexedCollection subclass: #FixedSizeCollection
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

В протоколе методов класса переопределяются все четыре сообщения из класса `Collection` для создания новых экземпляров. Приведем новую реализацию только одного из них.

```
with: firstObject
  with: secondObject
  with: thirdObject
  with: fourthObject
  | answer |
  answer := self new: 4.
  answer at: 1 put: firstObject.
  answer at: 2 put: secondObject.
  answer at: 3 put: thirdObject.
  answer at: 4 put: fourthObject.
  ^ answer
```

Поскольку размер экземпляра класса `FixedSizeCollection` фиксирован и не может изменяться, сообщение `add:` посылать таким объектам нельзя, поэтому в протоколе экземпляров этого класса метод для сообщения `add:` реализован как `^self invalidMessage`. Остальные сообщения экземплярам класса `FixedSizeCollection` имеют следующую реализацию.

```
collect: aBlock
  | answer index size |
  size := self size.
  answer := OrderedCollection new: size.
  index := 1.
```

```

[index > size]
  whileFalse: [
    answer add: (aBlock value: (self at: index)).
    index := index + 1].
^ (self species new: size)
  replaceFrom: 1
  to: size
  with: answer

```

```

copyReplaceFrom: start to: stop with: aCollection
| answer size1 size2 |
size1 := aCollection size.
size2 := self size + size1 - (stop - start + 1).
answer := self species new: size2.
answer
  replaceFrom: 1
  to: start - 1
  with: self
  startingAt: 1.
answer
  replaceFrom: start
  to: start + size1 - 1
  with: aCollection.
^ answer
  replaceFrom: start + size1
  to: size2
  with: self
  startingAt: stop + 1

```

```

deepCopy
  ^self shallowCopy

```

```

remove: anObject ifAbsent: aBlock
  ^self invalidMessage

```

```

select: aBlock
| answer index size anObject |
size := self size.
answer := OrderedCollection new: size.
index := 1.

```

```

[index > size]
  whileFalse: [
    anObject := self at: index.
    (aBlock value: anObject)
    ifTrue: [answer add: anObject].
    index := index + 1].
size := answer size.
^(self species new: size)
  replaceFrom: 1
  to: size
  with: answer

```

shallowCopy

```

| answer |
answer := self class new: self size.
answer replaceFrom: 1
  to: self size
  with: self
  startingAt: 1.
^ answer

```

size

```

“Вернуться к методу из класса Object”
<primitive: 62>

```

storeOn: aStream

```

| size index firstTime |
size := self size.
aStream
  nextPutAll: '(';
  nextPutAll: self class name;
  nextPutAll: ' new: '.
size printOn: aStream.
aStream nextPut: $).
index := 1.
firstTime := true.
[index <= size]
  whileTrue: [
    firstTime
    ifFalse: [aStream nextPut: $;].

```

```

    firstTime := false.
    aStream nextPutAll: 'at: '.
    index printOn: aStream.
    aStream nextPutAll: ' put: '.
    (self at: index) storeOn: aStream.
    index := index + 1].
firstTime
    ifFalse: [aStream nextPutAll: ';yourself'].
aStream nextPut: $)

```

4.1 Класс **Array**

Экземпляр класса **Array** (**Массив**) в качестве элементов может содержать любые объекты и представляет собой простейшую структуру для хранения объектов, имеющих целые внешние ключи-индексы. Класс **Array** полностью реализует протокол класса **FixedSizeCollection**, практически ничего нового в него не добавляя, а лишь переопределяя сообщения **printOn:** и **storeOn:**.

4.2 Класс **Interval**

Экземпляр класса **Interval** (будем называть его интервалом) представляет собой набор чисел, составляющих конечную арифметическую прогрессию. Напомним, что арифметическая прогрессия — это последовательность чисел, в которой задан первый член прогрессии, а каждый последующий член вычисляется прибавлением к предыдущему одного и того же числа, называемого разностью прогрессии. Прогрессия называется конечной, если она содержит конечное число членов. Чтобы определить все члены некоторой конечной арифметической прогрессии, можно указывать не число ее членов, а пределы (границы), в которых должны располагаться все члены прогрессии. Таким образом, интервал, например, может содержать все целые числа от 1 до 100, или же все четные числа от 2 до 100. Чуть более сложный пример: арифметическая прогрессия с положительными членами, начинающаяся со 100 и имеющая разностью число -20; легко видеть, что это последовательность чисел 100,80,60,40,20.

Класс **Interval** наследует сообщения из своих суперклассов и дополнительно он поддерживает сообщения для инициализации экземпляра класса **Interval** и для доступа к величинам, характеризующим данный

экземпляр (арифметическую прогрессию). Отметим, что все элементы экземпляра класса `Interval` создаются в момент инициализации экземпляра, поэтому элементы из экземпляра не могут быть удалены, и никакие новые элементы в экземпляр не могут быть добавлены.

Interval class protocol

`from: startInteger to: stopInteger` — возвращает экземпляр класса `Interval`, начинающийся со `startInteger` и заканчивающийся `stopInteger`, с разностью прогрессии равной 1.

`from: startInteger to: stopInteger by: stepInteger` — возвращает экземпляр класса `Interval`, начинающийся со `startInteger` и заканчивающийся `stopInteger`, с разностью прогрессии `stepInteger`.

В дополнение к сообщениям наследуемым из суперклассов, протокол экземпляра класса `Interval` поддерживает сообщение для определения разности арифметической прогрессии (сообщение `increment`).

Класс `Number` содержит два сообщения, позволяющие применять более короткую запись для создания экземпляров класса `Interval`. Это сообщения `to: stop` и `to: stop by: step`. Таким образом, выражение `Interval from: 1 to: 10` эквивалентно выражению `1 to: 10` в том смысле, что они оба создают интервал из целых чисел от 1 до 10 с шагом 1. Аналогично, создать экземпляр класса `Interval` положительных чисел с первым элементом равным 100 и разностью равной -20 (то есть арифметическую прогрессию 100, 80, 60, 40, 20) можно как с помощью выражения `Interval from: 100 to: 1 by: -20`, так и с помощью выражения `100 to: 1 by: -20`.

Интервал не обязательно должен состоять из целых чисел. Например, арифметическую прогрессию 10 10.2 10.4 10.6 10.8 11 11.2 ... 40 можно создать или с помощью выражения `Interval from: 10 to: 40 by: 0.2` или с помощью выражения `10 to: 40 by: 0.2`.

Сообщение `do:` применительно к интервалам обеспечивает функцию, аналогичную циклу в других языках программирования. Например, выражение языка **Algol**

```
for i:= 10 step 6 until 100 do
  begin
```

```
< предложения >  
end
```

в «переводе» на язык Smalltalk запишется выражением

```
(10 to: 100 by: 6) do: [:i | <предложения> ].
```

Экземпляры класса `Number` отвечают на сообщение `to:by:do:`, поэтому приведенное выше выражение можно записать и без круглых скобок, то есть в виде

```
10 to: 100 by: 6 do: [:i | <предложения> ].
```

Например, увеличить на единицу каждый 6-ой элемент из экземпляра класса `OrderedCollection` с именем `numbers`, можно так

```
6 to: numbers size  
  by: 6  
  do: [:index | numbers at: index put: (numbers at: index) + 1].
```

Это выражение создает арифметическую прогрессию 6 12 18 ... до числа не превосходящего размера набора `numbers`; если размер набора меньше 6, то ничего не происходит, иначе элементы набора с индексами 6, 12, 18 и т.д., до последнего возможного индекса, увеличиваются на 1.

Приведем полную реализацию класса `Interval`.

```
FixedSizeCollection subclass: #Interval  
  instanceVariableNames:  
    'beginning end increment '  
  classVariableNames: ''  
  poolDictionaries: ''
```

Interval class methods

```
from: beginningInteger to: endInteger  
  ^self basicNew  
    initBegin: beginningInteger  
    end: endInteger  
    incr: 1
```



```
from: beginningInteger
to: endInteger
by: incrementInteger
^self basicNew
    initBegin: beginningInteger
    end: endInteger
    incr: incrementInteger
```

Interval methods

```
at: anInteger
| answer |
anInteger > 0
    ifTrue: [
        answer := beginning + (increment * (anInteger - 1)).
        (increment < 0
            and: [answer between: end and: beginning])
            ifTrue: [^ answer].
        (increment > 0
            and: [answer between: beginning and: end])
            ifTrue: [^ answer]].
^self errorInBounds: anInteger
```

```
at: anInteger put: aNumber
^self invalidMessage
```

```
increment
^increment
```

```
initBegin: beginningInteger
end: endInteger
incr: incrementInteger
    "Private"
beginning := beginningInteger.
end := endInteger.
increment := incrementInteger.
increment = 0
    ifTrue: [^self error: 'interval increment is 0']
```

size

$\wedge 0$ max: end — beginning increment + 1

species

\wedge Array

4.3 Класс String

Экземпляр класса **String** (Строка) в качестве элементов может иметь только экземпляры класса **Character**. Экземпляры класса **String** понимают сообщения **at:**, **at:put:**, **size**, **replaceFrom:to:with:startingAt:** как и все индексированные наборы. Кроме того, класс **String** поддерживает дополнительный протокол сравнения своих экземпляров, подобный тому, что определен в классе **Magnitude**. Некоторые примеры, использующие строки, уже приводились ранее.

String instance protocol

- < aString** — возвращает **true**, если символы строки-приемника предшествуют символам строки-аргумента **aString**; не ASCII-символы игнорируются.
 - <= aString** — возвращает **true**, если символы строки-приемника предшествуют символам строки-аргумента **aString** или совпадают с ними; не ASCII-символы игнорируются.
 - > aString** — возвращает **true**, если символы строки-приемника следуют за символами строки-аргумента **aString**; не ASCII-символы игнорируются.
 - >= aString** — возвращает **true**, если символы строки-приемника следуют за символами строки-аргумента **aString** или совпадают с ними; не ASCII-символы игнорируются.
-

Все символы строки можно конвертировать в строку только строчных или только прописных символов. Еще строку можно преобразовать в экземпляр класса **Symbol**.

String instance protocol

asLowerCase — возвращает строку-приемник, состоящую из строчных символов.

asUpperCase — возвращает строку-приемник, состоящую из прописных символов.

asSymbol — возвращает экземпляр класса **Symbol**, чьи символы совпадают с символами строки-приемника.

Приведем примеры применения таких сообщений.

выражение	результат
'first string' asUpperCase	'FIRST STRING'
'First String' asLowerCase	'first string'
'First' asSyndol	First

В завершение раздела приведем реализации только первого и последнего сообщений из этого протокола, реализация второго сообщения аналогична первому.

asLowerCase

```
| answer size index aCharacter |
size := self size.
answer := String new: size.
index := 1.
[index <= size] whileTrue: [
    (aCharacter := self at: index) isUpperCase
    ifTrue: [aCharacter :=
        (aCharacter asciiValue + UpperToLower) asCharacter].
    answer at: index put: aCharacter.
    index := index + 1].
^ answer
```

asSymbol

```
^ Symbol intern: self
```

4.4 Класс `Symbol`

Экземпляр класса `Symbol` (**Имя**) — это уникальная последовательность символов. Класс `Symbol` (в отличие от класса `String`) не поддерживает сообщение `new`: и определяет в протоколе класса новые сообщения.

`Symbol` class protocol

`intern: aString` — возвращает экземпляр класса `Symbol`, чьи символы те же, что и в строке-аргументе `aString`.

`mustBeSymbol: anObject` — сообщает об ошибке, если аргумент `anObject` не является экземпляром класса `Symbol`.

Напомним, что кроме того экземпляр класса `Symbol` можно ввести литерально, используя символ `'#'` как префикс определяющей последовательности символов. Например, `#dave` — экземпляр класса `Symbol`, состоящий из 4-х символов. Но выводится на печать экземпляр класса `Symbol` без префикса.

Преимуществом экземпляров класса `Symbol` является то, что система заботится об их уникальности, и два литерально совпадающие экземпляра этого класса совпадают и как объекты, расположенные в памяти машины. Именно эта особенность отличает класс `Symbol` от класса `String`, в котором одинаковые строки могут храниться в памяти машины в нескольких физически различных экземплярах. Такой способ хранения позволяет вместо проверки равенства строк использовать более эффективную проверку совпадения адресов, по которым хранятся в памяти имена. С другой стороны, система запрещает манипуляции с внутренним содержанием имени. Для редактирования имя преобразуется в строку, над которой и выполняются необходимые преобразования, после чего строку можно включить в систему как новое имя.

Ввиду очевидности почти всех сообщений, посылаемых экземплярам классов `String` и `Symbol`, их реализацию здесь мы приводить не будем и предлагаем изучить ее самостоятельно, используя, например, браузер классов.

Литература

- [1] Goldberg A., Robson D., Smalltalk-80. *The language*. — Addison-Wesley Publishing Company, 1988.
- [2] Буч Г. *Объектно-ориентированное проектирование с примерами применения*. — М.: Конкорд, 1992.
- [3] Смолток. *Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3* — М.: Ин-т проблем информатики РАН, 1995
- [4] Иванов А., Кремер Ю., *Язык Smalltalk: концепция объектно-ориентированного программирования* // КомпьютерПресс — 1992, № 4 — С. 21–31
- [5] Фути К., Судзуки Н., *Языки программирования и схемотехника СБИС*: пер. с япон. — М.: «Мир», 1988
- [6] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Общие концепции и синтаксис*. — Ростов-на-Дону: УПЛ РГУ, 1995
- [7] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Интерфейс пользователя и среда программирования*. — Ростов-на-Дону: УПЛ РГУ, 1995

Содержание

1	Класс IndexedCollection	3
2	Класс OrderedCollection	7
2.1	Описание протокола	8
2.2	Реализация класса	9
3	Класс SortedCollection	15
3.1	Реализация класса	17
4	Класс FixedSizeCollection	18
4.1	Класс Array	22
4.2	Класс Interval	22
4.3	Класс String	26
4.4	Класс Symbol	28
	Литература	29