

Ю.А. Кирютенко, В.А. Савельев

Объектно-ориентированное программирование
и язык Smalltalk.

Класс Collection и его подклассы.

Часть 1

Ростов-на-Дону

1997

Объектно-ориентированное программирование
и язык Smalltalk.

Класс Collection и его подклассы.

Часть 1

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, интерфейсу пользователя и среде программирования. В двух ее частях описывается протокол и реализация 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 Иерархия класса Collection

Набор — это группа (совокупность, объединение) объектов. Эти объекты называются элементами набора. Например, массив — это набор. Так массив

```
#('word' 3 5 $G (1 2 3))
```

состоит из пяти элементов, первый — строка 'word', второй и третий — числа 3 и 5, четвертый — символ G, а пятый — массив из трех чисел. Элемент-строка 'word' — это тоже набор, в данном случае это набор, состоящий из четырех символов.

Наборы обеспечивают представление основных структур данных в среде Smalltalk. Рассмотрим сначала иерархию подклассов класса Collection. Кроме указанных классов в этой иерархии есть и другие, которые мы здесь рассматривать не будем.

Наборы, определяемые разными подклассами из этой иерархии, отличаются по своим свойствам. Одно из отличий состоит в том, определен или нет порядок среди элементов набора. Другое отличие состоит в том, имеет или нет набор фиксированный размер или он при необходимости может меняться. Существуют наборы, доступ к элементам которых происходит по известным вне набора именам (так называемым ключам). Вид ключей определяет еще одно отличие между классами наборов. Элементы одних наборов доступны по числовым натуральным индексам, которые определяют порядок элементов в наборе, а в других используется явно связанные с элементами набора внешние объекты, служащие ключами поиска.

Неотсортированный набор с внешними ключами — это экземпляр класса Dictionary. Его ключами могут быть любые объекты системы; сравнение ключей на совпадение проверяется с помощью равенства (=). Класс Dictionary имеет подкласс IdentityDictionary, чьи внешние ключи сравниваются на совпадение с помощью операции тождества (==). Очень часто ключами в этом классе выступают экземпляры класса Symbol. Элементы экземпляров классов Bag и Set не сортируются и не связываются ни с какими ключами. Дубликаты элементов возможны в экземплярах класса Bag, и невозможны в экземплярах класса Set.

имя класса	порядок	размер	дубли	ключи	элементы
Bad	нет	любой	да	нет	любой
IndexedCollection	да	о.п.	о.п.	целое	о.п.
FixedSizeCollection	да	фикс	о.п.	целое	любой
Array	да	фикс	да	целое	SmallInteger
Interval	да!	фикс	нет	целое	Number
String	да	фикс	да	целое	Character
Symbol	да	фикс	да	целое	Character
OrderedCollection	да	любой	да	целое	любой
SortedCollection	да!	любой	да	целое	любой
Set	нет	любой	нет	нет	любой
Dictionary	нет	любой	нет	любой	любой
IdentityDictionary	нет	любой	нет	любой	любой

*В таблице использованы следующие сокращения: **да!** — набор упорядочен в соответствии с внутренними характеристиками, **о.п.** — свойство определяется подклассом.*

Все упорядоченные наборы — экземпляры подклассов класса `IndexedCollection`. Во всех них, элементы доступны через ключи, которые являются целыми положительными индексами. Подклассы класса `IndexedCollection` поддерживают разные способы упорядочивания своих элементов. Еще одно различие между этими подклассами состоит в том, могут ли элементы быть произвольными объектами или они должны быть экземплярами определенного класса.

Порядок элементов является внешним для экземпляров классов `OrderedCollection`, `FixedSizeCollection`. Для `OrderedCollection` последовательность операций добавления и удаления элементов определяет порядок элементов. Элементами экземпляров класса `OrderedCollection` могут быть любые объекты. Подклассами класса `FixedSizeCollection` являются классы `Array`, `String`, `Interval`. Элементами экземпляров классов `Array` быть любые объекты, элементами экземпляров класса `String` могут быть только экземпляры класса `Character`, а элементами экземпляров класса `Interval` — только числа.

Порядок среди элементов экземпляров классов `Interval` и `SortedCollection` определяется внутренними свойствами самих элементов. Элементы экземпляра класса `Interval` составляют арифметическая прогрессия, которая однозначно определяется во время инициализации этого экземпляра. Для класса `SortedCollection` порядок элементов определяется задаваемым для данного экземпляра блоком сортировки. Наша

цель — описать подклассы класса `Collection` и все особенности их протоколов сообщений, а также привести простые поясняющие примеры.

2 Добавление, удаление, тестирование и перечисление в классе `Collection`

Прежде всего мы опишем протокол класса `Collection`, разделяемый всеми наборами. Сообщения в классе `Collection` реализуются или в самом общем виде, пригодном для любого вида набора, или как

`self implementedBySubclass.`

Напомним, что сообщения реализуется так, когда ассоциированный с ним метод зависит от представления принимающего сообщение объекта. Каждый подкласс должен переопределять такие сообщения, удовлетворяя все «требования подкласса». В целях эффективности, подклассы могут переопределять и другие сообщения класса, которые с новым методом использовать предпочтительнее. Поэтому каждое сообщение, описанное в этом разделе, понимается любым видом набора, за исключением тех, которые специально отменяют такое сообщение.

Подкласс может выполнять некоторые сообщения посредством выражения

`self invalidMessage`

результатом выполнения которого будет подсказка программисту о том, что данное посланное сообщение нельзя посылать экземплярам данного класса. Например, подобным образом в классе `Interval` реализовано сообщение с селектором `at:put:`, поскольку порядок среди элементов экземпляра этого класса определяется их внутренними свойствами и в него нельзя вмешиваться извне.

Класс `Collection` поддерживает 4 типа (категории) сообщений для доступа к элементам наборов

- сообщения добавления новых элементов
- сообщения удаления элементов
- сообщения проверки принадлежности элемента набору
- сообщения перечисления элементов

Протокол добавления, удаления и тестирования в классе `Collection` таков:

Collection instance protocol

`add: newObject` — включает объект `newObject` как один из элементов приемника; возвращает `newObject`.

`addAll: aCollection` — включает все элементы набора `aCollection` как элементы приемника; возвращает `aCollection`.

`remove: oldObject` — удаляет объект `oldObject` как элемент приемника; возвращает `oldObject`, если элемент удален; если в приемнике нет такого элемента, выдает сообщение об ошибке.

`remove: oldObject ifAbsent: anExceptionBlock` — удаляет объект `oldObject` как элемент приемника; если несколько элементов, равных `oldObject` содержатся в приемнике, удаляет первый найденный; возвращает `oldObject`, если элемент удален; если в приемнике нет такого элемента, выполняет блок `anExceptionBlock` и возвращает результат его вычисления.

`removeAll: aCollection` — удаляет все элементы аргумента `aCollection` из приемника; возвращает `aCollection`, если удаление всех элементов прошло успешно, иначе выдает сообщение об ошибке.

`includes: anObject` — возвращает `true`, когда хотя бы один из элементов приемника равен аргументу `anObject`.

`isEmpty` — возвращает `true`, когда в приемнике нет элементов.

`occurencesOf: anObject` — возвращает целое число равное числу вхождений элемента `anObject` в приемник.

Для того, чтобы продемонстрировать, как использовать эти сообщения, введем в рассмотрение набор `lotteryA`, равный `Bag(272 572 852 156)`, и набор `lotteryB`, равный `Bag(572 621 274)`. То есть это экземпляры класса `Bag`, подкласса класса `Collection`. Сам класс `Collection` — это абстрактный класс, который только описывает общий протокол для

всех своих подклассов. Класс `Collection` не достаточно конкретизирован, чтобы обеспечить хранение элементов набора, и потому не позволяет реализовать в себе все свои сообщения. Эта неполнота описания и не позволяет классу `Collection` создавать экземпляры. Класс `Bag` — уже конкретизированный подкласс класса `Collection` в том смысле, что он обеспечивает все необходимое для хранения элементов и реализует все сообщения, которые не были реализованы в его суперклассе.

Все наборы понимают сообщение `size`, которое возвращает количество элементов в наборе. Так мы можем определить, что `lotteryA size` есть 4, а `lotteryB size` есть 3. Вычисляя по порядку следующие сообщения, получим

выражение	результат	lotteryA, если изменилось
<code>lotteryA isEmpty</code>	false	
<code>lotteryA includes: 572</code>	true	
<code>lotteryA add: 596</code>	596	Bag(272 572 852 156 596)
<code>lotteryA addAll: lotteryB</code>	Bag(572 621 274)	Bag(272 274 852 156 596 572 572 621)
<code>lotteryA occurrencesOf: 572</code>	2	
<code>lotteryA remove: 572</code>	572	Bag(272 274 852 156 596 572 621)
<code>lotteryA size</code>	7	

Заметим, что сообщения `add:` и `remove:` возвращают свои аргументы, а не сам набор, так что можно получить доступ к вычисленному аргументу. Сообщение `remove: anObject` удаляет только один элемент из набора-приемника, равный аргументу `anObject`, а не все элементы набора равные ему.

Сообщение `remove: oldObject ifAbsent: anExceptionBlock` использует блок для того, чтобы определить поведение набора-приемника в том случае, когда удаляемый объект отсутствует в наборе. Если приемник сообщения не содержит элемента равного аргументу `oldObject`, выполняется блок-аргумент `anExceptionBlock`. Такой блок может, например, содержать код, обеспечивающий обработку ошибки или просто позволяющий проигнорировать ее. Так выражение

`lotteryA remove: 121 ifAbsent: []`

ничего не делает, если числа 121 не окажется в наборе `lotteryA`.

По умолчанию сообщение `remove: oldObject`, когда не находит элемента `oldObject` в наборе-приемнике, посылает набору сообщение `error: 'object is not in the collection'` ('объект в наборе отсутствует'). Напомним, что сообщение `error:` определено в классе `Object` и потому понимается всеми объектами системы.

Обратимся к той части протокола класса `Collection`, который включает сообщения перечисления элементов набора, предоставляя возможность перечислить все элементы набора и использовать каждый из них при вычислении блока. Основное сообщение перечисления — это сообщение `do: aBlock`. В нем аргументом является одно аргументный блок `aBlock`, который вычисляется, для каждого элемента из набора, получая его в качестве аргумента. Пусть, например, `letters` — некоторый набор символов, и мы хотим определить сколько раз в нем встречаются символы `a` и `A`.

```
count := 0.
```

```
letters do: [:each | each asLowercase == $a  
            ifTrue: [count := count + 1] ].
```

Здесь происходит увеличение счетчика `count` на 1 всякий раз, когда элемент из набора `letters` оказывается совпадающим либо с `a`, либо с `A`. Желаемый результат — последнее значение счетчика. Мы можем использовать `==` вместо `=`, так как все объекты класса `Character` уникальны.

Шесть основных сообщений перечисления определены в классе `Collection`. В описании этих сообщений говорится, что при их выполнении для сохранения результатов создается «новый набор, подобный приемнику сообщения». Эту фразу надо понимать в том смысле, что новый набор — экземпляр того же самого класса, что и набор-приемник сообщения. Например, если приемник сообщения `select: aBlock` является экземпляром класса `Set` или `Array`, то возвращаемый сообщением результат, является экземпляром класса `Set` или `Array`, соответственно. В системе `Smalltalk` на этот счет есть только одно исключение: в реализации класса `Interval`, в ответ на сообщение перечисления возвращается экземпляр класса `Array`, а не новый экземпляр класса `Interval`. Причина этого исключения в том, что элементы экземпляра класса `Interval` создаются в момент создания самого экземпляра, и потому невозможно сохранить какие бы то ни было элементы внутри уже существующего

интервала.

Collection instance protocol

do: aBlock — вычисляет аргумент **aBlock** для каждого элемента приемника.

select: aBlock — вычисляет аргумент **aBlock** для каждого элемента приемника; собирает в новый набор, подобный приемнику сообщения, только те элементы набора-приемника, для которых вычисление блока **aBlock** возвращает значение **true**; возвращает новый набор.

reject: aBlock — вычисляет аргумент **aBlock** для каждого элемента приемника; собирает в новый набор, подобный приемнику сообщения, только те элементы набора-приемника, для которых вычисление блока **aBlock** возвращает значение **false**; возвращает новый набор.

collect: aBlock — вычисляет аргумент **aBlock** для каждого элемента приемника; возвращает новый набор, подобный приемнику сообщения, куда включаются значения, возвращаемые блоком при каждом вычислении.

detect: aBlock — вычисляет аргумент **aBlock** для каждого элемента приемника; возвращает первый элемент, для которого вычисление блока возвращает значение **true**; если такого значения нет, сообщает об ошибке.

detect: aBlock ifNone: expectationBlock — вычисляет аргумент **aBlock** для каждого элемента приемника; возвращает первый элемент, для которого вычисление блока возвращает значение **true**. Если такого значения нет, вычисляет аргумент **expectationBlock**, который должен быть блоком без аргументов.

inject: thisValue into: binaryBlock — вычисляет аргумент **binaryBlock** снова для каждого элемента приемника; блок **binaryBlock** должен иметь два аргумента: второй аргумент блока — элемент из набора-приемника, первый аргумент — значение, возвращенное этим блоком при предыдущем вычислении, при этом стартовое значение первого аргумента — значение аргумента **thisValue**; возвращает последний результат вычисления блока-аргумента **binaryBlock**.

С помощью перечисляющих сообщений можно произвести выбор элементов из набора. Например, мы могли бы определить количество символов, совпадающих с `a` или `A` в наборе `letters`, используя сообщение `select`: следующим образом

```
(letters select: [:each | each asLowercase == $a]) size.
```

То есть, сначала создать новый набор, содержащий только те элементы набора-приемника, которые равны `a` или `A`, а затем вычислить размер этого набора.

Можно получить тот же самый результат, используя сообщение `reject`:

```
(letters reject: [:each |  
  each asLowercase ~~ $a]) size.
```

То есть, сначала создать новый набор, исключая из `letters` те элементы набора-приемника, которые не равны `a` или `A`, а затем вычислить размер этого набора.

При выборе между сообщениями `select`: и `reject`: следует опираться на наилучшее представление для тестирования элементов набора-приемника. Если выбор элементов лучше выражается в терминах принадлежности, то следует предпочесть сообщение `select`:, если же выбор элементов лучше выражается в терминах отбраковки неподходящих элементов, то следует предпочесть сообщение `reject`:.

Используя сообщение `inject:into:`, программист определяет имена временных переменных и избегает явной инициализации объекта, в котором накапливается результат. Например, в ранее приведенном примере подсчета числа символов в наборе `letters`, совпадающих с `a` или `A`, мы вводили счетчик `count`:

```
count := 0.  
letters do: [:each | each asLowercase == $a  
  ifTrue: [count := count + 1]].
```

Используя сообщение `inject:into:`, можно добиться того же. Как и раньше, мы вновь аккумулируем результат в переменной `count`, задавая начальное значение равным нулю, но это уже временная переменная. Когда следующий символ в наборе `letters` равен `a` или `A`, то к `count`

добавляется 1, иначе добавляется 0.

letters inject: 0

```
into: [:count :nextElement |
      count + (nextElement asLowercase == $a
        ifTrue: [1]
        ifFalse: [0])].
```

Еще один пример. Предположим, что множество **employees** — это набор элементов, представляющий работников некоторого предприятия, каждый из которых может отвечать на сообщение **salary**, возвращая его наибольший заработок. Создать множество тех работников, которые зарабатывают не менее 10.000, можно так

```
employees select: [:each | each salary >= 10000]
```

или так

```
employees reject: [:each | each salary < 10000].
```

Использование перечисляющих сообщений позволяет достаточно просто сформировать из имеющегося набора новый набор, элементы которого удовлетворяют некоторым условиям. Предположим необходимо составить набор, элементами которого должен быть заработок каждого рабочего из набора **employees**. Это можно сделать так:

```
employees collect: [:each | each salary].
```

Новый набор будет иметь тот же размер, что и набор **employees**. Каждый элемент нового набора — заработок соответствующего рабочего из набора **employees**.

Легко произвести и выборку из набора. Предположим надо найти хотя бы одного рабочего с заработком, превышающим 20.000. Выражение

```
employees detect: [:each | each salary > 20000]
```

вернет такой элемент из набора `employees`, если он есть, если же такого элемента нет, система сообщит об ошибке: `'object is not in the collection'` ('такого объекта в наборе нет'). Следующее выражение сделает ту же работу, но в случае, когда ни одного нужного элемента нет, вернет `nil`.

```
employees detect: [:each | each salary > 20000] ifNone: [nil].
```

3 Реализация протокола класса `Collection`

Наборы занимают значительное место в системе. И для того, чтобы лучше с ними познакомиться, и для того, чтобы получить представление о методах программирования в такой системе, мы дополнительно посмотрим, как реализованы в классе `Collection` соответствующие методы. Прежде всего отметим, что класс `Collection` не может обеспечить выполнение следующих сообщений из основного протокола, которые зависят от представления набора и реализуются подклассом:

```
add:anObject, remove: anObject ifAbsent: aBlock, do: aBlock.
```

Все остальные сообщения в основном протоколе реализуются с помощью этих сообщений.

3.1 Добавление элементов

Начнем с методов, которые позволяют вносить в набор новые элементы.

```
add: anObject  
  self implementedBySubclass  
  
addAll: aCollection  
  aCollection do: [ :each | self add: each].  
  ^ aCollection
```

Обратите внимание, что реализация сообщения с `addAll: aCollection` зависит от сообщений `do: aBlock` и `add: anObject`; порядок добавления элементов из набора-аргумента `aCollection`, зависит и от порядка в котором перечисляются элементы (`do:`) и от способа, в котором элементы включаются в набор (`add:`).

3.2 Удаление элементов

Сообщения `remove: anObject` и `removeAll: aCollection` определяются в терминах базисного сообщения с селектором `remove:ifAbsent:`, которое должно реализовываться в подклассах. Эти методы сообщают об ошибке, если элемент, который необходимо удалить отсутствует в наборе.

```
remove: anObject ifAbsent: exceptionBlock  
  ^ self implementedBySubclass
```

```
remove: anObject  
  ^ self remove: anObject ifAbsent: [self errorAbsentObject]
```

```
removeAll: aCollection  
  aCollection do: [ :each | self remove: each].  
  ^ aCollection
```

Здесь используется частное сообщение класса `Collection`, указывающее программисту на отсутствие удаляемого элемента в наборе-приемнике

```
errorAbsentObject  
  ^ self error: 'object is not in the collection'
```

В классе `Collection` реализовано еще одно сообщение об ошибке, которое информирует программиста о невозможности получить доступ к элементам набора по индексу (оно встретиться нам позже):

```
errorNotIndexable  
  ^ self error: self class name,  
    's do not respond to indexing messages'
```

3.3 Проверка состояния

Все сообщения протокола проверки состояния набора полностью реализованы в классе `Collection`.

```
isEmpty
```

```

^ self size = 0

includes: anObject
  self do: [:each | anObject = each ifTrue: [^ true]].
  ^ false

occurrencesOf: anObject
  ^ self inject: 0
    into: [:occurrences :each |
      occurrences + (anObject = each ifTrue: [1]
        ifFalse: [0])]

```

Реализация сообщения перечисления `inject:into:` будет приведена ниже. Заметим, что ответом на сообщение `includes: anObject` (как, впрочем, и на сообщение `isEmpty`) являются булевы объекты системы `true` или `false`.

Сообщение `size` (размер) наследуется из класса `Object`, но переопределяется частью подклассов класса `Collection`

3.4 Перечисление элементов

Обратимся к реализации методов перечисления.

```

do: aBlock
  self implementedBySubclass

collect: aBlock
  | newCollection |
  newCollection := self species new.
  self do: [ :each | newCollection add: (aBlock value: each)].
  ^ newCollection

detect: aBlock
  self detect: aBlock ifNone: [self errorAbsentObject]

detect: aBlock ifNone: exceptionBlock
  self do: [ :each | (aBlock value: each) ifTrue: [^ each]].
  ^ exceptionBlock value

```

```

inject: thisValue into: binaryBlock
  | nextValue |
  nextValue := thisValue.
  self do: [ :each | nextValue := binaryBlock
                    value: nextValue
                    value: each]

  ^ nextValue

reject: aBlock
  ^self select: [ :element | (aBlock value: element) not]

select: aBlock
  | newCollection |
  newCollection := self species new.
  self do: [ :each | (aBlock value: each)
                  ifTrue: [newCollection add: each]].
  ^ newCollection

```

В методах, связанных с сообщениями `collect: aBlock` и `reject: aBlock`, используется сообщение `species`, которое посылается псевдопеременной `self`. Сообщение `species` реализовано в классе `Object` и возвращает класс объекта-приемника.

```

species
  ^self class

```

Таким образом выражение `self species new` создает новый экземпляр того же самого класса, что и класс объекта-приемника. Это сообщение используется для создания “подобных” объектов при операциях преобразования/копирования сложных структур данных и если создание точного “подобия” объекта невозможно или нежелательно, этот метод можно переопределить, создавая объекты другого класса, допускающие операции, невозможные с оригиналом.

Мы уже отмечали, что для некоторых классов наборов (например, `Interval`) нельзя таким образом создать “подобный” экземпляр, поэтому эти классы наборов должны переопределять сообщение `species`.

Метод для сообщения `inject: thisValue into: binaryBlock` вычисляет двух аргументный блок `binaryBlock`, используя по одному разу каждый элемент из набора-приемника для своего второго аргумента. Каждый

предыдущий результат вычисления этого блока берется в качестве его первого аргумент в следующем вычислении, начальное значение первого аргумента определяется аргументом `thisValue`. Результат последнего вычисления блока `binaryBlock` возвращается как результат вычисления исходного сообщения.

Причина введения двух похожих сообщений `detect: aBlock` и `detect: aBlock ifNone: exceptionBlock` подобна той, по которой существуют два сообщения для удаления элементов `remove: anObject` и `remove: anObject ifAbsent: aBlock`. В более общем случае (`detect: aBlock`), выводиться сообщение об ошибке, если в наборе-приемнике нет элемента, для которого вычисление блока-аргумента `aBlock` отлично от `false`; программист может избежать этого уведомления об ошибке, используя второе сообщение и производя в этом случае (в случае ошибки) вычисление второго блока-аргумента `exceptionBlock`.

4 Создание новых наборов

В самом начале приводились примеры таких наборов, которые выражались как литералы. Это были экземпляры классов `Array` и `String`. Например, выражение для создания массива из строк имеет вид `#('first' 'second' 'third')`, где каждый элемент массива — это строка, задаваемая литерально.

Сообщения `new` и `new:` также можно использовать для создания новых наборов. Дополнительно, протокол класса для класса `Collection` содержит специальные сообщения, обеспечивающие создание нового набора с одним, двумя, тремя или четырьмя элементами. Эти сообщения обеспечивают краткую запись при создании тех видов наборов, которые не допускают литеральных представлений.

Collection class protocol

`with: anObject` — возвращает набор с одним элементом `anObject`.

`with: anObject1 with: anObject2` — возвращает набор с двумя элементами `anObject1` и `anObject2`.

`with: anObject1 with: anObject2 with: anObject3` — возвращает набор с тремя элементами `anObject1`, `anObject2` и `anObject3`.

with: anObject1 with: anObject2

with: anObject3 with: anObject4

— возвращает набор, содержащий четыре элемента anObject1, anObject2, anObject3 и anObject4.

Например, создать экземпляр класса **Set** с тремя элементами, которые являются символами **s**, **e**, **t**, можно, вычисляя выражение

Set with: \$s with: \$e with: \$t.

Заметим, что основная причина, по которой только эти четыре, а не больше и не меньше, сообщения создания экземпляров обеспечиваются протоколом класса, состоит в том, что именно такое их количество оказалось достаточным для решения всех тех задач, ради которых эти сообщения вставлялись разработчиками в систему.

Приведем реализацию только последнего из этих методов, которые, напомним, являются методами класса (реализация остальных — аналогична).

```
with: anObject1 with: anObject2
```

```
with: anObject3 with: anObject4
```

```
  ^ self new.
```

```
    add: anObject1;
```

```
    add: anObject2;
```

```
    add: anObject3;
```

```
    add: anObject4;
```

```
  yourself
```

Очевидно, что успешная реализация каждого из сообщений создания экземпляра зависит от способности вновь созданного экземпляра отвечать на сообщение **add:**.

5 Преобразование между классами наборов

Допустимые преобразования между различными видами наборов зависят от способа представления в системе экземпляра каждого подкласса класса **Collection**. В протоколе преобразования определено 5 сообщений, позволяющих представить приемник сообщения как экземпляр класса **Bag**, **Set**, **OrderedCollection** или **SortedCollection**. Эти сообщения

определены в классе `Collection` для того, чтобы обеспечить преобразование любого набора в экземпляр любого из указанных классов. Заметим, что упорядочивание элементов набора-приемника, чьи элементы не упорядочены, при преобразовании в набор упорядоченных элементов, производится произвольно.

Collection instance protocol

`asArray` — возвращает экземпляр класса `Array` с теми же самыми элементами, что и у набора-приемника.

`asBag` — возвращает экземпляр класса `Bag` с теми же самыми элементами, что и у набора-приемника.

`asSet` — возвращает экземпляр класса `Set` с теми же самыми элементами, что и у набора-приемника, при этом дубликаты уничтожаются.

`asOrderedCollection` — возвращает экземпляр класса `OrderedCollection` с теми же самыми элементами, что и у набора-приемника; упорядочивание элементов производится произвольно.

`asSortedCollection` — возвращает экземпляр класса `SortedCollection` с теми же самыми элементами, что и у набора-приемника, отсортированными так, что каждый предыдущий элемент меньше или равен последующему.

`asSortedCollection: aBlock` — возвращает экземпляр класса `SortedCollection` с теми же самыми элементами, что и у набора-приемника, отсортированными согласно аргументу `aBlock`.

Так например, если набор `lotteryA` - это экземпляр класса `Bag`, содержащий в качестве элементов числа 272, 572, 852, 156, 596, 272, 572, то набор `lotteryA asSet` содержит в качестве элементов числа 852, 596, 156, 572, 272, а набор `lotteryA asSortedCollection` содержит в качестве элементов те же самые числа, но отсортированные по возрастанию: 156, 272, 272, 572, 572, 596, 852.

Преобразование любого набора в экземпляр одного из классов `Bag`, `Set`, `OrderedCollection`, `SortedCollection` выполняется так: создается новый экземпляр нужного класса, куда затем добавляют нужные элементы из преобразуемого набора-приемника. В большинстве случаев,

новый экземпляр имеет тот же самый размер, что и первоначальный набор. В случае преобразования набора в экземпляр класса `OrderedCollection`, элементы добавляются в конец последовательности (`addLast:`), независимо от порядка перечисления элементов в источнике. Поскольку реализации преобразующих сообщений почти аналогично, приведем реализацию только двух из них.

`asArray`

```
| answer index |  
answer := Array new: self size.  
index := 1.  
self do: [:element | answer at: index put: element.  
    index := index + 1].  
^ answer
```

`asOrderedCollection`

```
^ (OrderedCollection new self size)  
    addAll: self;  
    yourself
```

Подводя итог сказанному, стоит еще раз подчеркнуть следующее. Любой набор можно преобразовать в экземпляр классов `Bag`, `Set`, `OrderedCollection`, `SortedCollection`. Любой индексированный набор можно преобразовать в экземпляр класса `Array`. Как мы увидим далее, экземпляры классов `String` и `Symbol` могут преобразовываться друг в друга. Никакие наборы не могут преобразовываться в экземпляры класса `Interval`.

6 Класс Bag

Класс `Bag` — подкласс класса `Collection` — описывает простейший вид наборов. Его экземпляр (сумка) представляет набор, чьи элементы не упорядочены и не имеют ключей, поэтому он не понимает сообщений `at:` и `at:put:` (они реализованы в нем посредством выражения `self errorNotIndexable`). В ответ на сообщение `size` экземпляр класса `Bag` возвращает число элементов набора. Экземпляр класса `Bag` ведет себя

согласно протокола для всех наборов. Общее описание класса не ограничивает числа одинаковых элементов (дублей) в экземпляре. Класс **Bag** подчеркивает это общее правило следующим специальным сообщением для добавления элементов.

Bag instance protocol

add: newObject withOccurrences: anInteger — добавляет аргумент **newObject** целое число раз **anInteger** в набор-приемник. Возвращает аргумент **newObject**.

Чтобы привести содержательный пример, предположим, что мы определили класс **Product**, экземпляры которого представляют наименование бакалейного товара и его цену. Пусть новый экземпляр класса **Product** создается посредством сообщения **of: aName at: aPrice**, посылаемого классу, а цену любого бакалейного товара можно определить посылая сообщение **price** соответствующему экземпляру. Теперь мы можем «наполнить необходимыми товарами нашу продовольственную сумку»:

```
sack := Bag new.  
sack add: (Product of: #steak at: 5.80)  
sack add: (Product of: #carrots at: 0.10) withOccurrences: 4.  
sack add: (Product of: #potatoes at: 0.50) withOccurrences: 6.  
sack add: (Product of: #milk at: 2.20).
```

Сумма затрат на приобретение этих товаров (которая получается равной 11.40) может быть определена вычислением выражений

```
amount := 0.  
sack do: [:eachProduct | amount := amount + eachProduct price]
```

или вычислением выражения

```
sack inject: 0  
into: [:amount :eachProduct | amount + eachProduct price].
```

Экземпляры класса **Bag** не упорядочены, и хотя сообщения перечисления ими поддерживаются, программисту не следует делать каких-либо предположений о порядке, в котором перечисляются элементы.

Представим часть протокола реализации класса **Bag** в котором опущены некоторые частные методы. Выполняемые этими частными методами действия очевидны из их имен, а реализация либо тривиальна, либо использует тонкости конкретной реализации системы **Smalltalk**.

```
Collection subclass: #Bag
instanceVariableNames: 'elements '
classVariableNames: ''
poolDictionaries: ''
```

Bag methods

```
includes: anObject
    elements at: anObject ifAbsent: [^ false].
    ^ true
```

```
occurrencesOf: anObject
    ^ elements at: anObject ifAbsent: [^ 0]
```

Добавить новый элемент в экземпляр класса **Bag** посредством сообщения **add: newObject** — означает добавить в переменную экземпляра **elements** ассоциативную пару с ключом **newObject** и значением 1; для этого используется сообщение **add: anObject withOccurrences: 1**. При удалении элемента проверяется число его вхождений в экземпляр, которое уменьшается на 1, если оно больше 1, если же это число равно 1, то соответствующая ассоциативная пара удаляется из словаря **elements**. В связи с этим, необходимо переопределить сообщение **size**.

```
add: anObject
    elements at: anObject
        put: (self occurrencesOf: anObject) + 1.
    ^ anObject
```

```
add: anObject withOccurrences: anInteger
    elements at: anObject
```

```
    put: (elements at: anObject ifAbsent: [0]) + anInteger.  
    ^ anObject
```

```
remove: anObject ifAbsent: aBlock  
    | occurrences |  
    (occurrences := elements  
     at: anObject  
     ifAbsent: [^ aBlock value]) = 1  
     ifTrue: [elements removeKey: anObject]  
     ifFalse: [  
         elements at: anObject  
         put: occurrences - 1].  
    ^ anObject
```

```
size  
    | answer |  
    answer := 0.  
    elements associationsDo: [ :element |  
                             answer := answer + element value].  
    ^ answer
```

Перечисление элементов из экземпляра класса **Bag** понимается как выбор каждой пары из словаря **elements** и вычисление нужного блока с ключом этой пары в качестве аргумента блока столько раз, каково число-значение этой пары (то есть сколько раз данный элемент-ключ входит в сумку). В соответствии с этим и определяется метод для **do: aBlock**. Остальные перечисляющие методы наследуются из класса **Collection**.

```
do: aBlock  
    | value |  
    elements associationsDo: [ :association |  
                             value := association value.  
                             [value > 0] whileTrue: [  
                                 aBlock value: association key.  
                                 value := value - 1]]
```

7 Класс Set

Экземпляр класса **Set** (множество) — набор, элементы которого не упорядочены и не имеют ключей. Экземпляр класса **Set** не отвечает на сообщения **at:** и **at:put:**. Класс **Set** во всем подобен классу **Bag**, исключая только то, что его экземпляры не могут содержать дубликатов, поэтому добавление нового элемента происходит только тогда, когда его нет в экземпляре.

Каждый экземпляр класса **Set** имеет переменную экземпляра с именем **elementCount**. Назначение этой переменной — хранить число элементов этого множества, что позволяет избегать неэффективного вычисления размера данного множества посредством подсчета всех элементов. Таким образом сообщения с селекторами **new**, **new:** и **size** переопределяются классом **Set**; первые два так, чтобы инициализировать переменную **elementCount**, а последний — чтобы ничего не вычисляя, просто вернуть значением переменной экземпляра **elementCount**.

```
Collection subclass: #Set
instanceVariableNames: 'elementCount contents '
classVariableNames: ''
poolDictionaries: ''
```

Set class methods

```
new
  ^self new: 4

new: anInteger
  ^super new initialize: ( 1 max: anInteger)
```

Set methods

```
size
  ^elementCount
```

В методе для сообщения **new: anInteger**, псевдопеременная **super** используется для того, чтобы избежать рекурсии. Частное сообщение класса **Set** **initialize: anInteger** определяет первоначальные значе-

ния переменных экземпляра (`elementCount := 0. contents := Array new: anInteger`).

В следующих методах экземпляра существенно использование частного сообщения `findElementIndex: anObject`, чтобы оптимизировать поиск и определить индекс, с которого надо начинать искать в наборе указанный элемент. Поиск продолжается до тех пор, пока или аргумент `anObject` не будет найден или поиск обнаружит свободное место вместо искомого объекта; возвращаемый ответ — индекс последней проверенной позиции.

`findElementIndex: anObject`

“Private.”

| index indexedObject lastIndex |

lastIndex := contents size.

index := (anObject hash) \\ lastIndex + 1.

[(indexedObject := contents at: index) = anObject]

whileFalse: [(indexedObject == nil)

ifTrue: [^index].

(index := index + 1) > lastIndex

ifTrue: [index := 1]].

^index

`includes: anObject`

^((contents at: (self findElementIndex: anObject)) == nil) not

`occurrencesOf: anObject`

(self includes: anObject)

ifTrue: [^1].

^0

Число вхождений любого элемента в экземпляр класса `Set` не больше 1. Поэтому методы для сообщений добавления, удаления и перечисления надо переопределять. При этом используются частные вспомогательные сообщения, которые либо производят преобразование множества, если для исходного множества требуемая операция невыполнима (`grow`, `adjustSize`), либо оптимизируют (`findElementIndex:`) либо подготавливают (`rehashFrom:`) правильное выполнение операций над множеством. Отметим, что для того, чтобы гарантировать правильную работу методов, основанных на методике хеширования, остающиеся эле-

менты должны уплотняться (`rehashFrom:`) всякий раз, когда элемент удаляется из набора.

```
add: anObject
  | index |
  anObject isNil
    ifTrue: [" nil не может быть элементом множества."
              ^anObject].
  self adjustSize.
  (contents at: (index := self findElementIndex: anObject)) isNil
    ifTrue: [elementCount := elementCount + 1.
              ^contents at: index put: anObject].
  ^anObject
```

```
remove: anObject ifAbsent: aBlock
  | index |
  index := self findElementIndex: anObject.
  (contents at: index) == nil
    ifTrue: [^aBlock value].
  contents at: index put: nil.
  elementCount := elementCount - 1.
  self rehashFrom: index.
  ^anObject
```

```
do: aBlock
  | index element |
  index := contents size.
  [index > 0]
    whileTrue: [ (element := contents at: index) == nil
                  ifFalse: [aBlock value: element].
                  index := index - 1]
```

8 Классы Dictionary и IdentityDictionary

Экземпляр класса `Dictionary` (словарь) — это набор ассоциативных пар — экземпляров класса `Association` — представляющих собой упорядоченную пару объектов, первый из которых называется ключом (`key`), а второй значением (`value`) ассоциативной пары. Другими словами,

экземпляры класса `Dictionary` — это наборы, чьи элементы не упорядочены, но имеют точно определенные имена (ключи). Таким образом, элементы словаря — произвольные объекты с внешними ключами. Такое представление об экземплярах класса `Dictionary` находит отражение и в протоколе сообщений этого класса. Сообщения `includes:`, `do:`, `occurencesOf:`, имеют дело со значениями (`value`) словарей (то есть эти сообщения используют объект `value` из каждой ассоциативной пары словаря), а не с ключи или самими пары, в то время как сообщения `at:`, `at:put:` обращаются к ключам ассоциативных пар словарей. Класс `Dictionary` использует технику хеширования при размещении элемента внутри словаря, аналогичную той, которую использует суперкласс `Set`, но хеширование производится по ключам ассоциативных пар, а не по самим ассоциативным парам.

8.1 Протокол класса `Dictionary`

Протокол класса `Dictionary` расширяется для взаимодействия как с самими парами, так и с их значениями путем добавления сообщений `associationAt:`, `keyAtValue:`. Для того, чтобы обеспечить дополнительный контроль во время поиска элементов словаря, определяется сообщение `at: key ifAbsent: aBlock`, с помощью которого программист может определить свои действия в тех случаях, когда нужный ключ не найден.

Dictionary instance protocol

`at: key ifAbsent: aBlock` — возвращает значение по заданному ключу `key`; если ключ не найден, возвращается результат выполнения блока `aBlock`.

`associationAt: key` — возвращает ассоциативную пару с заданным ключом `key`; если ключ не найден, сообщается об ошибке.

`associationAt: key ifAbsent: aBlock` — возвращает ассоциативную пару с заданным ключом `key`; если ключ не найден, возвращается результат выполнения блока `aBlock`.

`keyAtValue: value` — возвращает первый найденный ключ для значения `value`; если такого значения нет, возвращает `nil`.

keyAtValue: value ifAbsent: aBlock — возвращает первый ключ, найденный для значения **value**; если такого значения нет, возвращает результат вычисления блока **aBlock**.

keys — возвращает экземпляр класса **Set**, содержащий все ключи приемника.

values — возвращает экземпляр класса **Bag**, содержащий все значения приемника, включая все дубликаты.

Как пример использования класса **Dictionary**, создадим словарь с именем **opposites**, состоящий из слов-ключей и их антонимов-значений.

```
opposites := Dictionary new.  
opposites at: #hot put: #cold.  
opposites at: #push put: #pull.  
opposites at: #stop put: #go.  
opposites at: #come put: #go.
```

Еще один способ, позволяющий добавить в словарь новую ассоциативную пару, состоит в использовании сообщения с селектором **add:** и аргументом — экземпляром класса **Association**.

```
opposites add: (Association key: #front value: #back).  
opposites add: (Association key: #top value: #bottom).
```

Теперь словарь **opposites** состоит из следующих пар ключ–значение. Заметьте, что одно и то же значение может быть связано с любым числом ключей, в то время как все ключи обязаны быть уникальными.

Воспользуемся сообщениями тестирования и с их помощью проверим наличие в словаре **opposites** значений (но не ключей!).

ключ	значение
hot	cold
push	pull
stop	go
come	go
front	back
top	bottom

выражение	результат
<code>opposites size</code>	<code>6</code>
<code>opposites includes: #cold</code>	<code>true</code>
<code>opposites includes: #hot</code>	<code>false</code>
<code>opposites occurrencesOf: #go</code>	<code>2</code>
<code>opposites at: #stop put: #start</code>	<code>start</code>

Четвертый пример еще раз показывает неуникальность значений в словаре. А вот каждый ключ в словаре уникален и может появиться только один раз, поэтому в последнем примере для уже существовавшего в словаре ключа `stop` определяется новое значение `start` вместо 'старого' значения `go`.

Кроме сообщений, используемых для тестирования значений, в класс `Dictionary` добавлены сообщения для тестирования элементов-пар словарей и ключей этих пар.

Dictionary instance protocol

`includes: anAssociation` — возвращает `true`, когда словарь-приемник содержит ассоциативную пару, равную аргументу `anAssociation`, иначе возвращает `false`.

`includesKey: key` — возвращает `true`, когда словарь-приемник содержит ассоциативную пару с ключом, равным аргументу `key`, иначе возвращает `false`.

Применим эти методы к словарю `opposites`.

выражение	результат
<code>opposites includes: (Association key: #come value: #go)</code>	<code>true</code>
<code>opposites includesKey: #hot</code>	<code>true</code>

Аналогично, протокол удаляющих сообщений из класса `Collection` расширяется в классе `Dictionary` для того, чтобы обеспечить доступ и к элементам-парам и к их ключам. Однако заметим, что сообщение `remove:` не используется классом `Dictionary`, поскольку при удалении элементов обязательна ссылка на ключ.

removeAssociation: *anAssociation* — удаляет из набора-приемника пару, равную аргументу *anAssociation*, и возвращает этот аргумент.

removeKey: *key* — удаляет из набора-приемника пару с ключом, равным аргументу *key*, возвращает значение, связанное с этим ключом. Если пары с указанным ключом в приемнике нет, сообщает об ошибке.

removeKey: *key* **ifAbsent:** *aBlock* — удаляет из набора-приемника пару с ключом, равным аргументу *key*, возвращает значение, связанное с этим ключом. Если пары с указанным ключом в приемнике нет, возвращает результат вычисления блока-аргумента *aBlock*.

Приведем примеры. При вычислении выражения

```
opposites removeAssociation: (Association key: #top
                             value: #bottom)
```

будет возвращен экземпляр класса *Association* с ключом *top* и значением *bottom*, а набор *opposites* станет имеет на 1 элемент меньше. При вычислении выражения

```
opposites removeKey: #hot
```

будет возвращен экземпляр класса *Association* с ключом *hot* и значением *cold*, а набор *opposites* станет имеет еще на 1 элемент меньше. При вычислении выражения

```
opposites removeKey: #cold
ifAbsent: [opposites at: #cold put: #hot]
```

произойдет следующее: поскольку пары с ключом *cold* в словаре нет, будет выполнен блок и словарь *opposites* станет иметь на 1 элемент больше.

Сообщение **do:** вычисляет свой блок-аргумент для каждого значения из словаря. Протокол перечисляющих сообщений, наследуемый

из класса `Collection`, в классе `Dictionary` вновь расширяется для того, чтобы включить в него сообщения перечисления для работы с ассоциативными парами и их ключами.

Dictionary instance protocol

`associationsDo: aBlock` — вычисляет аргумент `aBlock` для каждой пары из словаря-приемника.

`keysDo: aBlock` — вычисляет аргумент `aBlock` для каждого ключа из словаря-приемника.

Таким образом мы имеем различные возможности перечисления словарей. Для примера, предположим, что экземпляр класса `Set` — набор `newWord` — состоит из слов, которые должен выучить ученик. Пусть слова из словаря `opposites`, как ключи, так и значения, уже ученику известны. Вычисление следующих двух выражений удалит уже известные слова из набора `newWord` (первое выражение удалит значения, а второе — ключи).

```
opposites do: [:word | newWord remove: word ifAbsent: [ ] ].
opposites keysDo: [:word | newWord remove: word ifAbsent: [ ] ].
```

Тот же результат получится, если использовать сообщение перечисления для ассоциативных пар:

```
opposites associationsDo: [:each |
    newWord remove: each key ifAbsent: [ ].
    newWord remove: each value ifAbsent: [ ] ]
```

Сообщения доступа `keys` и `values` можно использовать, чтобы, например, получить соответствующие наборы слов из словаря `opposites`. Предположим, что все предыдущие примеры для словаря `opposites` выполнены, тогда

выражение	результат
opposites keys	Set('push' 'come' 'front' 'stop' 'cold')
opposites values	Bag('pull' 'go' 'back' 'start' 'hot')

8.2 Реализация класса Dictionary

Приведем теперь реализацию некоторых методов класса Dictionary. Отметим, что много работы в протоколе класса Dictionary выполняется частными сообщениями, которые либо наследуются из класса Set, либо подобны сообщениям, оптимизирующим поиск (findKeyIndex:).

```
Set subclass: #Dictionary
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Dictionary methods

```
add: anAssociation
  | index element |
  index := self findKeyIndex: anAssociation key.
  (element := contents at: index) == nil
    ifTrue: ["такой пары нет, сохранить ее"
      elementCount := elementCount + 1.
      contents at: index put: anAssociation]
    ifFalse: ["пара с таким ключом есть, заменить значение"
      element value: anAssociation value].
  self adjustSize.
  ^ anAssociation

associationAt: aKey
  ^self associationAt: aKey
    ifAbsent: [self errorAbsentKey]

associationAt: aKey ifAbsent: aBlock
  | answer |
  ^ (answer := self lookUpKey: aKey) == nil
```

```
ifTrue: [aBlock value]
ifFalse: [answer]
```

```
associationsDo: aBlock
  super do: aBlock
```

Именно `super do: aBlock`, поскольку сообщение `do: aBlock` переопределяется в классе `Dictionary` следующим образом

```
do: aBlock
  super do: [ :association |
    aBlock value: association value]
```

Реализация сообщений добавления, доступа и удаления тоже имеет в классе `Dictionary` свою “специфику”.

```
at: aKey
  | answer |
  ^ (answer := self lookUpKey: aKey) == nil
    ifTrue: [self errorAbsentKey]
    ifFalse: [answer value]
```

```
at: aKey ifAbsent: aBlock
  | answer |
  ^ (answer := self lookUpKey: aKey) == nil
    ifTrue: [aBlock value]
    ifFalse: [answer value]
```

```
at: aKey put: anObject
  self add: (Association key: aKey value: anObject).
  ^ anObject
```

```
includes: anAssociaton
  self do: [ :element | element = anAssociaton
    ifTrue: [^ true]].
  ^ false
```

```
remove: anObject ifAbsent: aBlock
  ^ self invalidMessage
```



```

removeAssociation: anAssociation
    self removeKey: anAssociation key

removeKey: aKey
    self removeKey: aKey ifAbsent: [self errorAbsentKey].

removeKey: aKey ifAbsent: aBlock
    | index |
    index := self findKeyIndex: aKey.
    (contents at: index) == nil
        ifTrue: [^ aBlock value].
    contents at: index put: nil.
    elementCount := elementCount - 1.
    self rehashFrom: index.
    ^ aKey

```

Обратите внимание на симметрию между `at:put:` и `add:`. Различие состоит только в действии, если элемент не найден — в случае `at:put:` создается и сохраняется в словаре новая ассоциативная пара; а в случае `add:`, сам аргумент `anAssociation` сохраняется как новая ассоциативная пара словаря.

Сообщение `select: aBlock` переопределяется, чтобы выбирать ассоциативные пары из словаря-приемника по результату выполнения аргумента `aBlock` с параметром, равным значению ассоциативной пары.

```

select: aBlock
    | answer |
    answer := self species new.
    self associationsDo: [ :each |
        (aBlock value: each value)
            ifTrue: [answer add: each]].
    ^ answer

```

8.3 Класс IdentityDictionary

Класс `IdentityDictionary`, подкласс класса `Dictionary`, почти все сообщения своего суперкласса переопределяет так, чтобы производить про-

верку идентичности ключей вместо проверки равенства ключей. Поэтому в классе `IdentityDictionary` операции вставки и поиска является более эффективными, чем в классе `Dictionary`, и в тех случаях, когда ключи уникальны, всегда лучше использовать экземпляры класса `IdentityDictionary`.

Реализацию всех этих сообщений мы здесь приводить не будем. Их, как и остальные сообщения класса `Dictionary`, читатель может изучить, используя браузер иерархии классов системы `Smalltalk`.

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	Иерархия класса Collection	3
2	Добавление, удаление, тестирование и перечисление в классе Collection	5
3	Реализация протокола класса Collection	12
3.1	Добавление элементов	12
3.2	Удаление элементов	13
3.3	Проверка состояния	13
3.4	Перечисление элементов	14
4	Создание новых наборов	16
5	Преобразование между классами наборов	17
6	Класс Bag	19
7	Класс Set	23
8	Классы Dictionary и IdentityDictionary	25
8.1	Протокол класса Dictionary	26
8.2	Реализация класса Dictionary	31
8.3	Класс IdentityDictionary	33
	Литература	35