

Ю.А. Кирютенко, В.А. Савельев

Объектно-ориентированное программирование  
и язык Smalltalk.  
Потоки и файлы.

Ростов-на-Дону

1997

# Объектно-ориентированное программирование и язык Smalltalk.

Потоки и файлы.

## Аннотация

Методическая разработка посвящена современному направлению в программировании — объектно-ориентированной методологии программирования и языку Smalltalk и является продолжением вышедших ранее методических разработок по объектно-ориентированному программированию и языку Smalltalk. В данной работе описываются протоколы и реализация класса Stream (Поток) и его подклассов. Рассматриваются классы, поддерживающие работу с файловой системой DOS и экраном. Всюду далее мы имеем ввиду из реализации в языке Smalltalk/V фирмы Digitalk (США). Те, кто прочитал наши первые методички по общим концепциям, синтаксису языка, интерфейсу пользователя и имеет в своем распоряжении его русифицированную версию могут пользоваться ей, по ходу чтения соотнося английские названия с соответствующими русскими именами. Как нам кажется, после первоначального знакомства с основными концепциями и синтаксисом, языковой барьер уже не выглядит непреодолимым. А как показала практика, англоязычный вариант языка более доступен, чем его русифицированная версия.

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 № 7 от 30 июня 1997 года.

Настоящие методические указания набраны в системе L<sup>A</sup>T<sub>E</sub>X с использованием кириллических шрифтов семейства LH (дизайнеры О. Лапко и А. Ходулев).

# 1 Протокол класса Stream

Классы наборов обеспечивают систему Smalltalk основными структурами данных для совместного хранения объектов как в нелинейных так и в линейных группах. Протокол этих классов обеспечивает непосредственный доступ к элементам наборов (их хранение и восстановление). Через сообщения перечисления, поддерживается вызов всех элементов по-порядку. Однако, не поддерживается протоколом наборов обращение к индивидуальным элементам, одному за одно обращение.

Если нет легко вычисляемого внешнего имени для каждого элемента набора, то непрерывное перечисление индивидуальных элементов не может выполняться эффективно. Поэтому один из возможных подходов при перечислении подразумевает, что набор некоторым образом сам помнит, какой элемент был доступен (использован) последним. Мы называем это запоминаемое положение *указателем позиции* или *позиционным указателем*. Возможность разделяемого доступа к последовательности элементов, однако, означает что необходимо создать и поддерживать отдельную, внешнюю память для последнего использованного элемента.

Класс **Stream** представляет возможность поддерживать позиционный указатель на элементы из набора объектов. Мы используем фразу 'поток над набором', понимая под этим доступность элементов набора таким образом, что можно производить перечисление или сохранение каждого элемента, одного за одно обращение, возможно, смешивая эти операции. Создавая несколько потоков над одним и тем же набором, возможно поддерживать и несколько позиционных указателей.

Есть несколько способов поддерживать позиционный указатель для потока над набором. Самый общий подход состоит в использовании целого числа как индекса. Этот подход может использоваться для любого набора, чьи элементы имеют индекс. Класс **IndexedCollectons** и его подклассы относятся к этой категории и в основном такие потоки и представляется в системе Smalltalk экземплярами подклассов класса **Stream**. При этом потоки поддерживают позиционный указатель как индекс, который увеличивается каждый раз, когда поток обращается к элементу. Второй способ поддерживать позиционный указатель состоит в том, чтобы использовать генератор для образования объектов. Пример этого вида потока в языке Smalltalk — класс **Random**.

## 1.1 Класс Stream

Определение класса **Stream** в системе **Smalltalk** следующее

```
Object subclass: #Stream
  instanceVariableNames: 'collection position readLimit '
  classVariableNames: ' '
  poolDictionaries: 'CharacterConstants '
```

Класс **Stream** — абстрактный подкласс класса **Object**, порождающий следующую иерархию. Как суперкласс, класс **Stream** определяет общий протокол для потока над набором. прежде всего в этот протокол включаются сообщения для чтения и записи (сохранения) элементов в наборе, хотя не все подклассы класса **Stream** могут поддерживать эти два вида операций. В системе **Smalltalk** класс **Stream** и его подклассы используются для доступа к файлам, устройствам и внутренним объектам как к последовательности символов или других объектов. Три переменные экземпляра, определяемые в классе **Stream** имеют следующий смысл

```
Stream
  ReadStream
    Random
  WriteStream
    ReadWriteStream
      FileStream
      TerminalStream
```

**collection** — содержит экземпляр класса **IndexedCollection**, над которым открыт поток. Например, для файлового потока (экземпляра класса **FileStream**) эта переменная содержит строку — страницу файлового буфера.

**position** — содержит целое число, представляющее текущую позицию потока.

**readLimit** — содержит целое число, представляющее текущее число элементов в потоке.

Разделяемый классом словарь пула **CharacterConstants** определяет общие и наиболее часто используемые символьные константы (например, **Space** — для символа пробела, **Lf** — для символа пропуска строки, **Tab** — для символа табуляции и т.д.)

Экземпляры классов-потокa могут создаваться посредством посылки классу сообщения `on: aCollection`, в котором аргумент `aCollection` является набором доступных создаваемому потоку элементов. Но, опять же, не все классы создают свои экземпляры именно так. Реализация соответствующего метода класса такова

`on: anIndexedCollection`

```
"Возвращает экземпляр класса—приемника (новый
поток) над набором anIndexedCollection."
^(self new) setCollection: anIndexedCollection;
  setLimits
```

В этом методе используются два частных сообщения `setCollection:` и `setLimits`, первое из которых устанавливает переменную экземпляра `collection` равной аргументу сообщения, а второе переопределяется каждым подклассом и инициализируют все остальные переменные создаваемого экземпляра.

Следующий протокол содержит базисные сообщения чтения в потоках, основное из них — `next`.

### Stream instance protocol

---

`next` — возвращает следующий доступный элемент из приемника сообщения и сдвигает указатель на следующий доступный элемент.

`next: anInteger` — возвращает следующие `anInteger` доступных элементов из приемника сообщения; обычно, это набор того же самого класса, что и тот, который доступен потоку-приемнику, и сдвигает указатель на следующий доступный элемент.

`nextMatchFor: anObject` — читает следующий элемент из потока и определяет равен ли он аргументу `anObject`, при этом сдвигает указатель на следующий доступный элемент.

`contents` — возвращает все элементы из набора, доступного потоку-приемнику; обычно, это набор того же самого класса, что и тот, который доступен потоку-приемнику.

---

Базисное сообщение записи — `nextPut: anObject`. Многие сообщения из протоколов записи и чтения используют его в своих реализациях.

## Stream instance protocol

---

`nextPut: anObject` — сохраняет аргумент `anObject`, как следующий элемент доступный потоку-приемнику; возвращает `anObject`.

`nextPutAll: aCollection` — сохраняет элементы из аргумента `aCollection`, как следующие элементы доступные приемнику; возвращает `aCollection`.

`next: anInteger put: anObject` — сохраняет аргумент `anObject`, как следующий элемент доступный потоку-приемнику `anInteger` раз; возвращает `anObject`.

---

Класс `Stream` — абстрактный класс только потому, что он не обеспечивает реализацию сообщений `next` и `nextPut: anObject`. Реализация этих сообщений производится в его подклассах. Реализация некоторых сообщений, использующая эти не реализуемые классом `Stream` сообщения почти очевидна.

```
next: anInteger  
  | aStream |  
  aStream := WriteStream on: (collection species new: anInteger).  
  anInteger timesRepeat: [aStream nextPut: self next].  
  ^ aStream contents
```

```
next: anInteger put: anObject  
  anInteger timesRepeat: [self nextPut: anObject].  
  ^ anObject
```

```
nextMatchFor: anObject  
  ^ (self next) = anObject
```

Сообщения чтения и записи определяют может ли следующий элемент быть прочитан или записан и, если нет, сообщают об ошибке. Чтобы определить возможен ли доступ, программист может послать потоку сообщение `atEnd`. Непрерывное чтение элементов, которые используются как аргументы блока, может быть выполнено с помощью сообщения `do: aBlock`, подобно тому, как это происходило в классах наборов.

## Stream instance protocol

---

`atEnd` — сообщает имеет ли поток-приемник доступ к какому-либо объекту.

`do: aBlock` — вычисляет блок `aBlock` для каждого доступного элемента потока-приемника.

---

Реализация сообщения перечисления основывается на сообщениях `atEnd` и `next` и реализуется так:

```
do: aBlock  
[self atEnd] whileFalse: [aBlock value: self next].
```

Каждый подкласс класса `Stream` должен определять свои сообщения создания экземпляра. Отметим, что поток не может создаваться просто посылкой классу сообщения `new`, потому что поток должен быть проинформирован о том, какой набор ему доступен и каков начальный позиционный указатель.

Вот простой пример. Предположим, что доступный потоку с именем `accessor` набор — экземпляр класса `Array`. Пусть элементами массива будут экземпляры класса `Symbol`

Bob Dave Earl Frank Harold Jim Kim Mike Peter Rick Sam Tom.

И, наконец, предположим, что позиционный указатель таков, что `Bob` — следующий доступный элемент. Тогда

---

выражение

результат

---

accessor next	Bob
accessor next	Dave
accessor nextMatchFor: #Bob	false
accessor nextMatchFor: #Frank	true
accessor next	Harold
accessor nextPut: #James	James
accessor contents	(Bob Dave Earl Frank Harold James Kim Mike Peter Rick Sam Tom)
accessor nextPutAll: #(Karl Larry Paul)	(Karl Larry Paul)
accessor contents	(Bob Dave Earl Frank Harold James Karl Larry Paul Rick Sam Tom)
accessor next: 2 put: #John	John
accessor contents	(Bob Dave Earl Frank Harold James Karl Larry Paul John John Tom)
accessor next	Tom
accessor atEnd	true

---

Класс **Stream** поддерживает дополнительный протокол доступа к элементам набора и тестирования содержимого набора. Появляется возможность управлять производимым выбором.

### Stream instance protocol

---

**isEmpty** — возвращает **true**, если набор, доступный приемнику сообщения не имеет элементов, иначе возвращает **false**.

**peek** — возвращает следующий элемент набора (как и сообщение **next**), но не изменяет позиционный указатель (подсматривает следующий элемент); возвращает **nil**, если позиционный указатель в конце потока.

**peekFor: anObject** — определяет ответ как и сообщение **peek**, при этом, если следующий элемент равен **anObject**, тогда позиционный указатель увеличивается и возвращается **true**, иначе возвращается **false** и позиционный указатель не меняется.



`upTo: anObject` — возвращает набор элементов доступных потоку-приемнику, начинающийся со следующего доступного элемента и до, но не включая его, элемента равного `anObject`, если же `anObject` отсутствует в наборе, то до конца набора; позиционный указатель устанавливается на элемент, находящийся за `anObject`.

---

Реализация этих сообщений (кроме первого) приведена ниже. Последнее сообщение этого протокола использует примитивный метод системы.

```
peek
| anObject |
self atEnd
  ifTrue: [^ nil]
  ifFalse: [anObject := self next.
            self position: self position - 1.
            ^ anObject]

peekFor: anObject
self atEnd
  ifTrue: [^ false]
  ifFalse: [anObject = self next
            ifTrue: [^ true]
            ifFalse: [self position: self position - 1.
                       ^ false]].

upTo: anObject
< primitive : 82 >
^ self
  copyFrom: self position + 1
  to: ((self skipTo: anObject)
       ifTrue: [self position - 1]
       ifFalse: [self position])
```

Так как экземпляру класса **Stream** 'известно' как хранится позиционный указатель, протокол этого класса поддерживает перепределение позиционного указателя, установку позиционного указателя в начало, в конец, или в любую другую позицию доступного

набора.

## Stream instance protocol

---

**position** — возвращает текущий позиционный указатель потока-приемника.

**position: anInteger** — устанавливает текущий позиционный указатель потока-приемника для вызова следующего элемента на позицию **anInteger**. Если аргумент выходит за допустимые границы, сообщает об ошибке.

**reset** — устанавливает текущий позиционный указатель потока-приемника в начало набора.

**setToEnd** — устанавливает текущий позиционный указатель приемника сообщения в конец набора.

**skip: anInteger** — устанавливает текущий позиционный указатель потока-приемника в позицию равную сумме текущей позиции и аргумента **anInteger**, возможно подгоняя результат так, чтобы оставаться в допустимых границах.

---

Обратите внимание, что при реализации этих сообщений в одних случаях **readLimit** (например, в реализации сообщения **position:**) — переменная экземпляра, а в других (например, в реализации сообщения **setToEnd**) — частный метод, который возвращает значение этой переменной экземпляра.

```
position
  ^ position
```

```
position: anInteger
  (0 <= anInteger and: [anInteger <= readLimit])
    ifFalse: [^self error: 'position outside of stream'].
  position := anInteger
```

```
reset
  self position: 0
```

```
setToEnd  
    self position: self readLimit
```

## 1.2 Класс ReadStream

Класс **ReadStream** — конкретный подкласс класса **Stream**, который предоставляет доступ к элементам набора только для чтения. Ниже приведены примеры, показывающие как в этом случае использовать протокол, предоставляемый классом **Stream** и наследуемый всеми его подклассами. Определение самого класса **ReadStream**, которое ниже приведено полностью, мало что добавляет нового по сравнению со своим суперклассом, фактически впервые определяя сообщение `next`, которое не реализовано в классе **Stream**.

```
Stream subclass: #ReadStream  
    instanceVariableNames: ''  
    classVariableNames: ''  
    poolDictionaries: ''
```

ReadStream methods

```
contents  
    ^collection copyFrom: 1 to: readLimit
```

```
next  
    <primitive: 65>  
    position < readLimit  
        ifTrue: [ position := position + 1.  
                  ^collection at: position]  
        ifFalse: [^self error: 'read beyond end of stream']
```

```
setLimits  
    "Частный метод инициализации переменных экземпляра."  
    position := 0.  
    readLimit := collection size
```

Конечно, сообщения `nextPut:`, `next:put:`, `nextPutAll:` не должны посылаться экземплярам класса `ReadStream`, поскольку это вызовет только сообщение об ошибке. Итак, пусть

```
accessor := ReadStream on: #(Bob Dave Earl Frank Harold Jim
                           Kim Mike Peter Rick Sam Tom).
```

выражение	результат
<code>accessor next</code>	Bob
<code>accessor nextMatchFor: #Dave</code>	true
<code>accessor peek</code>	Earl
<code>accessor next</code>	Earl
<code>accessor peekFor: #Frank</code>	true
<code>accessor next</code>	Haroid
<code>accessor upTo: #Rick</code>	(Jim Kim Mike Peter)
<code>accessor position</code>	10
<code>accessor skip: 1</code>	the accessor itself
<code>accessor next</code>	Tom
<code>accessor atEnd</code>	true
<code>accessor reset</code>	the accessor itself

### 1.3 Класс `WriteStream`

Класс `WriteStream` — подкласс `Stream`, предоставляющий доступ к набору только для записи элементов в набор. Поэтому в определении этого класса дополнительно вводится еще одна переменная экземпляра `writeLimit`, которая содержит целое число, представляющее собой последнюю допустимую позицию для записи в наборе, на котором открыт поток-получатель. Кроме того, класс `WriteStream`, как и класс `Stream`, получает доступ к пулу `CharacterConstants`, хранящему символные константы системы.

```
Stream subclass: #WriteStream
  instanceVariableNames: 'writeLimit '
  classVariableNames: ' '
  poolDictionaries: 'CharacterConstants '
```

Именно в этом классе реализованы сообщения `nextPut:` и `nextPutAll:`.

```
nextPut: anObject
| index |
<primitive: 66>
index := position + 1.
index > writeLimit
    ifTrue: [self grow. writeLimit := collection size].
position := index.
collection at: index put: anObject.
^ anObject
```

```
nextPutAll: aCollection
| stopIndex |
stopIndex := position + aCollection size.
stopIndex <= writeLimit
    ifTrue: [collection
        replaceFrom: position + 1
        to: stopIndex
        with: aCollection
        startingAt: 1.
        position := stopIndex.
        ^ aCollection].
^ aCollection do: [ :element | self nextPut: element]
```

Ни одно из сообщений с селекторами `next`, `next:`, `do:` не должно посылаться экземплярам класса `WriteStream`, поскольку это вызовет только сообщение об ошибке.

Экземпляры класса `WriteStream` используются системой `Small-talk` как составная часть методов для печати или хранения строк описания любого объекта. Каждый объект в системе может отвечать на сообщения `printOn: aStream` и `storeOn: aStream`. Методы, реализующие эти сообщения, состоят из последовательности сообщений к аргументу, который является экземпляром одного из подклассов класса `Stream` и позволяет записывать элементы в доступный ему поток. Среди них: сообщение `nextPut: aCharacter`, сообщение `nextPutAll:` с аргументом — экземпляром класса `String` или `Symbol`. Ниже на примере мы проиллюстрируем эту идею.

Протокол печати класса **Object** включает сообщение **printString**. Реализация этого сообщения такова

```
printString  
  | aStream |  
  aStream := WriteStream on: (String new: 16).  
  self printOn: aStream.  
  ^ aStream contents
```

Если, например, набору послать сообщение **printString**, то ответом на него будет экземпляр класса **String** (строка), который и будет строкой-описанием экземпляра. Метод вначале создаст экземпляр класса **WriteStream**, который может сохранять набор в себе, затем пошлет набору сообщение **printOn:** с созданным потоком в качестве аргумента и, наконец, возвратит содержимое окончательно сформированного экземпляра класса **WriteStream**. Сообщение **storeString**, также понимаемое любым объектом системы, аналогичным образом реализовано в классе **Object**, отличие от **printString** состоит только в том, что второе выражение соответствующего метода содержит сообщения **storeOn: aStream** вместо сообщения **printOn: aStream**.

Общее правило, по которому наборы печатают свою строку-описание, состоит в том, чтобы сначала напечатать имя класса, которому принадлежит набор, затем левую круглую скобку, затем напечатать строку-описание каждого элемента набора, отделяя эти описания друг от друга пробелами, и завершить все это правой круглой скобкой. Так если набор является экземпляром класса **Set**, содержит четыре элемента **#one**, **#two**, **#three**, **#four** (четыре экземпляра класса **Symbol**), то этот набор будет введен в поток в виде

Set(one two three four)

Экземпляр класса **OrderedCollection** с теми же самыми элементами будет напечатан на поток в виде

OrderedCollection(one two three four)

Напомним, что любая подходящая строка-описание может быть результатом выполнения сообщения печати, но строка-описание, создаваемая сообщением **storeOn: aStream**, должна представлять “пра-

вильно построенное выражение”, вычисление которого позволяло бы восстановить объект, который описывался. Как пример, приведем реализация в классе `Collection` сообщения печати.

```
printOn: aStream
  aStream nextPutAll: self class name.
  aStream space.
  aStream nextPut: $(.
  self do: [:element | element printOn: aStream.
           aStream space].
  aStream nextPut: $)
```

Обратите внимание, что в теле этого метода, экземпляру `aStream` класса `WriteStream` посылается сообщение `space`. Это и ряд других сообщений определяются в классе `WriteStream`, чтобы поддерживать краткие выражения для разделителей в таких потоках. Вот они.

#### WriteStream instance protocol

---

`cr` — сохраняет символ возврата каретки как следующий элемент потока-приемника.

`space` — сохраняет пробел как следующий элемент в приемнике сообщения.

`tab` — сохраняет символ табуляции как следующий элемент в приемнике сообщения.

---

Таким образом, чтобы создать строку вида

'name	city
bob	New York
joe	Chicago
bill	Rochester'

для двух соответствующих массивов, надо вычислить выражения

```
| names cities aStream |
```

```

names := #(bob joe bill).
cities := #('New York' 'Chicago' 'Rochester').
aStream := WriteStream on: (String new: 16).
aStream nextPutAll: 'name'. aStream tab.
aStream nextPutAll: 'city'. aStream cr; cr.
names with: cities do:
    [ :name :city |
        aStream nextPutAll: name. aStream tab.
        aStream nextPutAll: city. aStream cr].
^ aStream contents.

```

## 1.4 Класс ReadWriteStream

Класс `ReadWriteStream` — подкласс `WriteStream`, экземпляры которого поддерживают как протокол класса `ReadStream` так и протокол класса `WriteStream`, поэтому в доступный такому потоку набор можно как записывать информацию так и читать информацию, содержащуюся в нем. Все методы для записи информации в поток этот класс наследует из класса `WriteStream`, а все методы для чтения информации из потока приходится определять заново, повторяя реализацию класса `ReadStream`. Это недостаток используемой в системе иерархии классов, не позволяющей множественного наследования, то есть такой ситуации, когда класс может иметь более одного непосредственного суперкласса.

## 2 Потоки генерируемых элементов

Второй способ, упомянутый в самом начале изложения, состоит в том, чтобы определить источник (генератор) чисел, посредством которого создаются следующие элементы набора. Этот вид потока разрешает производить только чтение элементов, но не запись.

Класс `Random` является подклассом класса `ReadStream`. Он определяет свои элементы основываясь на алгоритме, использующем некоторое начальное число как основу для определения всех последующих чисел. Поэтому класс `Random` обеспечивает свою реализацию сообщений `next` и `atEnd`. Процесс образования новых элементов теоретически бесконечен, в силу этого, экземпляры класса `Random`



не отвечают на сообщение `contents`; и хотя класс `Random` поддерживает сообщение `do:`, которое наследуются из класса `Stream`, метод, ассоциированный с ним, начавшись никогда не закончится без целенаправленного вмешательства программиста.

Приведем полную реализацию класса `Random`.

```
ReadStream subclass: #Random
instanceVariableNames: 'seed '
classVariableNames: ' '
poolDictionaries: ' '
```

Random class methods

```
new
^self basicNew setSeed
```

Random methods

```
atEnd
^false
```

```
next
| temp |
"Линейный конгруэнтный метод Лемера. Возвращает
псевдо—случайное число из интервала (0, 1)."
[seed := 13849 + (27181 * seed) bitAnd: 8r177777.
temp := seed / 65536.0. temp = 0] whileTrue: [ ].
^temp
```

```
setSeed
"Для определения псевдослучайного начального значения,
взять время из системных часов компьютера. Это — — —
большое положительное целое число, поэтому использовать
только младшие 16 битов."
seed := Time millisecondClockValue bitAnd: 8r177777
```

### 3 Класс FileStream

Класс `FileStream` (Файловый Поток) — подкласс класса `ReadStream`. Класс `FileStream` используется в системе `Smalltalk` как основа, на которой создается и функционирует файловая система. Все обращения к внешним файлам выполняются, используя экземпляр класса `FileStream`, который действует так, как будто ему доступна большая последовательность байтов или символов; элементы этой последовательности предполагаются целыми числами или символами (экземплярами классов `Integer` или `Character`).

Файловые потоки допускают использование всех ранее описанных методов. Помимо этого файловые потоки, используя объект класса `File` (Файл), позволяют организовать произвольный постраничный доступ к файлам DOS (с размером страницы в 512 байт). В свою очередь, для чтения и записи страниц файлов DOS, файлы используют экземпляры класса `FileHandle` (ОписательФайла). Класс `Directory` (Каталог) предоставляет возможности манипулирования каталогами дисков DOS. Далее будет представлено краткое описание упомянутых выше классов, связанных с доступом к файлам DOS. Класс `FileStream` определяется в системе следующим образом

```
ReadStream subclass: #FileStream
  instanceVariableNames: 'file pageStart writtenOn
                          lastByte lineDelimiter '
  classVariableNames: ' '
  poolDictionaries: ' CharacterConstants '
```

Как следует из определения, в дополнение к структуре, определяемой классами `Stream` и `WriteStream`, класс `FileStream` вводит новые переменные экземпляра, которые имеют следующий смысл

`file` — содержит экземпляр класса `File`, представляемый данным потоком.

`lastByte` — содержит наибольшую допустимую позицию в файле; для файловых потоков переменная `writeLimit` содержит наибольшую допустимую позицию в текущем буфере.

`lineDelimiter` — содержит один из символов: `Cr` — возврат каретки (carriage-return) или `Lf` — перевод строки (line-feed); строки

файлов либо разделяются парой CrLf или только одним символом Lf.

**pageStart** — содержит позицию текущего буфера относительно начала файла; позиция в экземпляре класса **FileStream** определяется как сумма позиции **pageStart** и **position** (смотрите ниже реализацию сообщений **position** и **position:**).

**writtenOn** — содержит логическое значение (**false** или **true**), определяющее были ли изменения в буфере, содержащем текущую страницу.

Файловые потоки создаются путем послыки сообщения классу **File**, которое частично или полностью задает путь доступа к файлу. Другим способом создания файлового потока является послыка объекту класса **Directory** сообщения, задающего имя конкретного файла в каталоге.

Рассмотрим примеры сообщений, посылаемых классу **File**:

```
File pathName: 'c:\smalltalk\chapter.1'  
File pathName: 'smalltalk\chapter.1'
```

В первом выражении полностью специфицирован путь доступа к файлу. Во втором выражении путь доступа специфицирован не полностью. В таких случаях недостающие части спецификации пути извлекаются из глобальной переменной **Disk**, которая связана с объектом-каталогом, содержащим информацию о каталоге из которого был запущен **Smalltalk/V**. Во втором выражении глобальная переменная **Disk** используется для получения полной спецификации пути доступа к указанному файлу.

Рассмотрим примеры создания файловых потоков путем послыки сообщений объекту-каталогу:

```
Disk file: 'chapter.1'  
Disk newFile: 'jink.fil'
```

В приведенных выше выражениях, аргументом может быть только имя файла. Спецификация пути доступа к файлу не допускается. Различие в двух сообщениях — **file:** и **newFile:**, заключается в том, что при использовании второго сообщения, ранее существовавший

файл, имеющий имя, совпадающее с аргументом, удаляется из каталога. Затем создается новый файл. Оба сообщения создают файл в каталоге, если он не существовал ранее.

Несколько замечаний по поводу файловой системы DOS. DOS не осуществляет автоматического обновления информации о файле в каталоге при записи в файл. Для обновления информации о файле существует два сообщения, посылаемых файловым потокам: `close` и `flush`.

Различие между этими сообщениями заключается в том, что первое (`close`) закрывает файловый поток, делая невозможным доступ к файлу DOS с помощью файлового потока-получателя сообщения. Второе сообщение (`flush`) вынуждает систему действительно произвести все операции записи, очистив буфера, и обновить информацию о файле в каталоге диска, файловый поток-получатель доступен для последующей работы с файлом. Заметим, что унаследованное Smalltalk'ом из операционной среды рабочих станций сообщение `flush`, в среде DOS моделируется с помощью запоминания позиционного указателя, закрытия файла и последующего его открытия с восстановлением позиционного указателя.

Файловые потоки распознают два различных разделителя строк — пару символов `CrLf`, принятую в текстовых файлах DOS или только символ `Lf`, принятый в ОС UNIX. При открытии файлового потока на существующем файле, разделитель строки определяется путем сканирования информации из начала файла. Новые файлы всегда получают разделитель строки в формате DOS. Приведенные ниже три выражения иллюстрируют возможности проверки и изменения формата разделителя строки в файлах:

```
stream lineDelimiter      "Возвращает разделитель — — — CrLf или Lf."
stream lineDelimiter: Lf  "Разделитель в Unix-формате Lf."
stream lineDelimiter: Cr  "Разделитель в DOS-формате CrLf."
```

Наиболее эффективный способ считывания информации из файлового потока — посылка ему сообщения `next`; а записи в файловый поток — посылка ему сообщения `nextPutAll`.

Завершая разговор о разделителях, приведем текст метода, предназначенного для преобразования текстовых файлов DOS в формат ОС UNIX.

```

| input output |
input := Disk file: 'chapter.dos'.
output := Disk newFile: 'stripped.unx'.
output lineDelimiter: Lf. "разделитель строки в формате UNIX"
[input atEnd]
    whileFalse: [output nextPutAll: input nextLine;
                "считывает очередную строку из файла"
                cr].
output close

```

Для того, чтобы лучше представлять работу указанных выше методов из протокола класса **FileStream**, приведем их реализацию. Все приводимые методы — методы экземпляра.

```

atEnd
    "Возвращает true, если позиционный указатель в конце потока
    (за последним элементом), иначе возвращает false."
    (position < readLimit or:
     [(pageStart + position) <= self updateLastByte])
    ifTrue: [^ false] ifFalse: [^ true]

```

```

close
    "Закрывает файл, ассоциированный с потоком—приемником,
    после записи в него всех необходимых данных."
    self readLimit.
    self writePage.
    file close

```

```

flush
    "Гарантирует, что на диск запишется вся информация,
    посланная потоку—приемнику."
    | position |
    position := self position.
    self setToEnd;
        readLimit;
        writePage.
    file flush.
    self position: position

```

lineDelimiter

"Возвращает символ—разделитель для файла  
из потока—приемника: или Cr или Lf."

^ lineDelimiter

lineDelimiter: aCharacter

"Заменяет символ—разделитель на символ aCharacter."

lineDelimiter := aCharacter

next

"Возвращает следующий символ, доступный потоку—приемнику  
и увеличивает позиционный указатель; сообщает об ошибке, если  
позиционный указатель расположен в конце потока."

<primitive: 65>

self position: self position.

self atEnd

ifTrue: [^ self error: 'read beyond end of stream']

ifFalse: [(position := position + 1) > readLimit

ifTrue: [self error: 'position outside of stream'].

^ collection at: position]

nextLine

"Возвращает строку символов потока—приемника  
до следующего символа—разделителя."

| line |

line := self upTo: lineDelimiter.

lineDelimiter == Cr

ifTrue: [self peekFor: Lf].

^ line

nextPut: aCharacter

"Записать символ aCharacter в поток—приемник."

writtenOn := true.

^ self primitiveNextPut: aCharacter

nextPutAll: aCollection

"Записать каждый из символов набора  
aCollection в поток—приемник."

| currentPosition stopIndex |

stopIndex := position + aCollection size.

```

stopIndex <= writeLimit
  ifTrue: [collection replaceFrom: position + 1
              to: stopIndex
              with: aCollection
              startingAt: 1.
            position := stopIndex.
            writtenOn := true.
            ^ aCollection].
^ aCollection do: [ :element | self nextPut: element]

```

pathName

```

" Возвратить полный путь файла,
  над которым создан поток—приемник."
| path |
path := file directory pathName.
path = ' \ '
  ifTrue: [path := ''].
^ (String with: file directory drive), ': ',
  path, ' \ ', file name

```

position

```

" Возвращает текущую позицию в потоке—приемнике."
^ pageStart + position - 1.

```

position: anInteger

```

" Устанавливает позиционный указатель
  равным числу anInteger."
self readLimit.
((anInteger + 1) < pageStart
 or: [(pageStart + collection size) <= (anInteger + 1)])
  ifTrue: [ self writePage.
pageStart := ((anInteger // collection size)
              * collection size) + 1.
readLimit := file readBuffer: collection
              atPosition: pageStart].
position := anInteger - pageStart + 1.
self updateLastByte

```

updateLastByte

```

" Возвращает позицию последнего байта в файле

```

и обновляет значение переменной экземпляра lastByte.”  
^ lastByte := lastByte max: position + pageStart - 1

writePage

”Если были изменения, записывает в файл  
из потока—приемника текущую страницу.”

writtenOn

ifTrue: [file writeBuffer: collection  
          ofSize: readLimit  
          atPosition: pageStart.  
writtenOn := false]

## 4 Класс Directory

Класс Directory (Каталог) предоставляет средства доступа к каталогам файловой системы DOS.

Object subclass: #Directory

instanceVariableNames: 'drive pathName volumeLabel '

classVariableNames: ' '

poolDictionaries: ' '

Именованные переменные экземпляра этого класса имеют следующий смысл

**drive** — содержит имя (экземпляр класса **Character**) дискового устройства.

**pathName** — содержит строку (экземпляр класса **String**), представляющую путь к каталогу-приемнику (начиная с корневого каталога), исключая имя дискового устройства.

**volumeLabel** — содержит строку, представляющую метку диска, на котором находится каталог.

Исходный вариант среды **Smalltalk** содержит три глобальных переменных, связанных с каталогами:

Disk, DiskA, DiskB



Глобальная переменная системы Disk содержит объект-каталог, из которой была запущена среда Smalltalk. Переменные DiskA и DiskB содержат корневые каталоги дисковых устройств A: и B:, соответственно. С помощью приведенных ниже примеров сообщений можно создать произвольные объекты-каталоги:

```
SampleDir := Directory new drive: $a; pathName: '\dirname'.
SampleDir := DiskA subDirectory: 'dirname'.
DiskC      := Directory new drive: $c; pathName: '\'
```

Необходимо отметить, что создание объекта-каталога не означает создание каталога на диске. Для создания каталога на диске объекту-каталогу необходимо послать сообщение **create**. Например:

SampleDir create

Объектам-каталогам можно посылать сообщения, возвращающие все подкаталоги данного каталога, все файлы в каталоге, а также сообщения создающие новый файл в каталоге, новый подкаталог и тому подобное. Метод класса **create:** для создания каталога DOS на диске и метод класса **pathName:** для создания нового каталога, реализованы следующим образом

**create:** newPathName

```
“Создать DOS—каталог на диске
с полным именем пути newPathName.”
| aDos old |
aDos := Dos new.
old := newPathName asAsciiZ.
aDos setReg: 6 to: old; setReg: 3 to: old;
    setRegHigh: 0 to: 16r39; interrupt: 16r21.
Dos checkDosError: aDos registers
```

**pathName:** aString

```
“Возвращает новый экземпляр класса Directory, с полным
описанием имени пути, содержащимся в аргументе aString.”
^self new drive: (aString at: 1);
    pathName: (aString copyFrom: 3 to: aString size)
```

Используемое в методе `create` сообщение `asAsciiZ` из протокола экземпляра класса `String`, посланное строке, возвращает эту строку с символом 0 в конце строки. В этом же сообщении используется экземпляр класса `Dos`, который эмулирует структуру регистров системы процессора Intel и вызов операционной системы через прерывания. Немного об этом мы расскажем в следующих разделах.

В завершение этого раздела приведем реализацию тех сообщений из протокола экземпляра класса `Directory`, которые использовались в этом и предыдущем разделах.

`create`

```
“Создает DOS—каталог на том же диске, что и каталог—приемник.”  
Directory create: (String with: self drive), ':', self pathName
```

`drive`

```
“Возвращает букву дискового устройства приемника.”  
^ drive
```

`drive: aCharacter`

```
“Аргумент aCharacter используется для инициализации  
устройства для каталога—приемника.”  
volumeLabel := String new.  
drive := aCharacter
```

`file: aString`

```
“Возвращает экземпляр класса FileStream  
для файла в текущем каталоге с именем aString;  
если такого файла не существует, создать его.”  
^ FileStream on: (File open: aString in: self)
```

`newFile: aString`

```
“Возвращает экземпляр класса FileStream  
для файла в текущем каталоге с именем aString;  
если такой файл существует сначала удалить его,  
а затем создать новый файл с таким именем.”  
| answer |  
answer := self file: aString.  
answer size > 0  
  ifTrue: [ File remove: answer pathName.
```

```
answer := self file: aString].  
^ answer
```

Подробное описание и реализацию остальных методов, содержащихся в классе `Directory` можно получить изучая этот класс с помощью, например, `ClassHierarchyBrowser`.

## 5 Классы `File` и `FileHandle`

Класс `File` (Файл) определяется следующим образом

```
Object subclass: #File  
  instanceVariableNames: 'name fileld directory '  
  classVariableNames: ' PageSize '  
  poolDictionaries: ' '
```

и обеспечивает поддержку последовательного или произвольного постраничного доступа к файлам DOS, необходимую для реализации файловых потоков. Каждая операция чтения позволяет считать страницу (512 байт — значение, хранящееся в переменной класса `PageSize`) файла, за исключением может быть последней страницы файла, которая может содержать меньше 512 байт. Одна операция записи позволяет записывать информацию размером от 1 байта до размера страницы. Класс `File` организует постраничный доступ, ориентированный на DOS, используя экземпляр класса `FileHandle` (УправлениеФайлом), который хранится в переменной экземпляра `fileld` и обеспечивает самый низкий уровень доступа к файлам. (Экземпляр класса `FileHandle` является просто массивом байт, структура которого определяется DOS). Значения, хранящиеся в двух оставшихся переменных экземпляра класса `File` очевидны, переменная `directory` содержит экземпляр класса `Directory`, представляющий каталог, в котором хранится файл, а переменная `name` содержит строку (экземпляр класса `String`), являющуюся именем файла.

Как правило, программист имеет дело с этими классами только тогда, когда возникает необходимость в разработке нового метода доступа к файлам, отличного от файлового потока.

Класс `File` среди методов класса содержит ряд полезных сообщений, позволяющих копировать, переименовывать и удалять файлы, часть из которых мы использовали в предыдущем изложении.

## File instance protocol

---

`changeModeOf: aString to: attrString` — изменяет атрибуты файла с именем `aString` на те, которые определены аргументом `attrString`. Последний аргумент может включать в себя символы: `$r` — только чтение, `$h` — скрытый, `$s` — системный, `$a` — архивный.

`copy: oldFile to: newFile` — копирует файл с именем `oldFile` в файл с именем `newFile`.

`open: aString in: aDirectory` — возвращает экземпляр класса `File` открытый на файле с именем `aString` в каталоге `aDirectory`.

`pathName: aString` — возвращает файловый поток с полным именем, заданным аргументом.

`pathName: aString in: aDirectory` — возвращает файловый поток с полным именем, заданным аргументами.

`remove: aString` — удаляет (стирает) файл с именем `aString`.

`rename: oldString to: newString` — переименовывает файл с именем `oldString` в файл с именем `newString`.

---

Ранее мы использовали и некоторые сообщения из протокола экземпляра класса `File`. Их описание и описание некоторых других полезных сообщений этого протокола приведены ниже.

## File instance protocol

---

`close` — закрыть файл-приемник.

`directory` — возвращает строку с именем каталога, содержащего файл-приемник.

`name` — возвращает строку, содержащую имя файла-приемника.

`open` — открыть файл-приемник.

`getDate` — возвращает массив, содержащий время и дату файла-приемника в формате DOS.

`setDate: anArray` — устанавливает время и дату файла-приемника в формате DOS, используя данные из аргумента `anArray`.

`size` — возвращает размер файла-приемника в байтах.

---

## 6 Класс `TerminalStream`

Класс `TerminalStream` (ПотокТерминала) является подклассом класса `ReadStream` и обеспечивает поддержку как ввода информации с клавиатуры или мыши так и вывода информации на экран терминала.

```
ReadStream subclass: #TerminalStream
  instanceVariableNames: 'state mouseOffset mouseTime '
  classVariableNames: ' '
  poolDictionaries: 'FunctionKeys CharacterConstants '
```

Определяемые переменные экземпляра имеют следующий смысл

`mouseOffset` — хранит положение курсора в момент нажатия правой кнопки мыши; при скроллинге эта позиция используется в качестве начальной позиции скроллинга.

`mouseTime` — содержит время в миллисекундах нажатия правой клавиши мыши; переменная используется для того, чтобы вычислять задержку между нажатием и освобождением клавиши мыши.

`state` — содержит имя метода, который должен выполняться в следующем состоянии, когда активизируется метод `read`.

С пулом `CharacterConstants` мы уже встречались раньше, пул `FunctionKeys` аналогичен пулу `CharacterConstants` и содержит переменные для кодов функциональных клавиш (класса `Character`), вводимых с клавиатуры или мыши.

Во всех методах системы **Smalltalk** для поддержки терминального ввода/вывода используется глобальная переменная **Terminal**, которая содержит экземпляр класса **TerminalStream**.

В классе **TerminalStream**, сообщение **next**, наследуемое из класса **Stream**, заменено на сообщение **read**. Сообщение **read** используется для считывания символов и кодов функциональных клавиш как от клавиатуры, так и от мыши. Совместно с сообщением **read** используются две глобальные переменные **FunctionKeys** и **MouseEvent**. Эти переменные принимают значения истина (**true**) или ложь (**false**) для указания источника информации, перед ее считыванием. Если информация поступает от мыши, то перед возвратом функционального кода мыши, глобальная переменная **MouseEvent** принимает значение **true**. Аналогично, при нажатии функциональной клавиши на клавиатуре, перед возвратом кода функциональной клавиши, переменная **FunctionKeys** принимает значение **true**. Нажатие на клавиатуре нефункциональных клавиш приводит к тому, что перед возвратом считываемого с клавиатуры символа обе переменные принимают значение **false**.

Сообщение **nextPut**: предназначено для вывода символа на экран терминала, но используется очень редко, только в особых случаях отладки. Вывод на экран **Smalltalk** осуществляет обычно с помощью посылки сообщений объектам класса **BitBlt** или его подкласса **CharacterScanner**. При этом экран терминала рассматривается как объект-форма (экземпляр класса **DisplayScreen**). Это позволяет копировать прямоугольные области экрана во внутренние формы (экземпляры класса **Form**) и наоборот. Перечисленные здесь классы обеспечивают работу с графическими объектами. Это достаточно большая группа классов, изучение которых не является предметом изучения в этой работе. Мы надеемся посвятить графическим классам отдельную методическую разработку.

Метод **read** использует примитив чтения (метод, написанный на ассемблере), который в свою очередь, для чтения с клавиатуры использует прерывание BIOS 16H. BIOS возвращает скан-код и ASCII-код нажатой клавиши, каждый из кодов представляется одним байтом. Для некоторых клавиш (например для функциональных), ASCII-код клавиши равен нулю. В этом случае примитив чтения при первом обращении возвращает символ-префикс равный нулю, и при втором — скан-код клавиши. Если ASCII-код клавиши не равен нулю, и клавиша не принадлежит дополнительной части клавиатуры (не в диапазоне скан-кодов от 71 до 83), примитив чте-

ния просто возвращает ASCII-код клавиши, без генерации символа-префикса. Если нажатая клавиша находится в дополнительной части клавиатуры, то примитив чтения, при первом обращении возвращает символ-префикс равный нулю, а при втором обращении — увеличенный на 90 ASCII-код клавиши.

В случае мыши, примитив чтения всегда при первом обращении возвращает символ-префикс. При перемещении мыши из одного положения в другое возвращается префикс с ASCII-кодом 255. Последующие два обращения к примитиву чтения возвращают x-координату и y-координату курсора. Координаты курсора возвращаются не в виде символов, а в виде коротких целых чисел. При нажатии или отпуске какой-либо кнопки мыши, возвращается символ-префикс с ASCII-кодом 254. Последующее обращение к примитиву чтения возвращает целое число, отражающее состояние кнопки (нажата или отпущена), а также факт нажатия кнопки мыши при нажатой клавише SHIFT. Перед возвратом из метода `read` это целое число преобразуется в функциональный код мыши.

Для обработки событий, связанных с мышью, драйвер мыши постоянно сообщает примитиву чтения о всех событиях, связанных с мышью, извлекая их из очереди, и используя стратегию “первый пришел, первый обслуживается”. Необходимо отметить, что последовательные перемещения мыши в очереди событий представляются как одно событие. События связанные с мышью имеют более высокий приоритет, чем события связанные с нажатием клавиш на клавиатуре.

Примитив чтения всегда возвращает объект-символ, за исключением x- и y- координат курсора и состояний нажатия и отпускания кнопок мыши. Приведенные ниже таблицы сводит в единое целое информацию, возвращаемую примитивом чтения. `Shift on` (клавиша SHIFT нажата) и `shift off` (клавиша SHIFT не нажата) отражают состояние клавиши SHIFT в момент возникновения какого-либо события (от клавиатуры или от мыши).

---

Кнопка мыши	Символ-префикс	shift on	shift off
<hr/>			
Нажатие левой кнопки	254	16r0101	16r1101
Отпускание левой кнопки	254	16r0001	16r0001
Нажатие правой кнопки	254	16r0102	16r1102
Отпускание правой кнопки	254	16r0002	16r0002

Нажатие средней кнопки	254	16r0103	16r1103
Отпускание средней кнопки	254	16r0003	16r0003
Перемещение мыши	255	х и у	х и у

---

Клавиша клавиатуры	Символ-префикс	shift on	shift off
Дополнительная клавиатура	0	скан-код	ASCII-код+90
Функциональные клавиши	0	скан-код	скан-код
Остальные клавиши	отсутствует	ASCII-код	отсутствует

---

## 7 Класс Dos

Связь тех перечисленных выше классов, которые взаимодействуют с системой DOS и аппаратурой IBM-совместимого персонального компьютера, осуществляется через экземпляр класса **Dos**, который в системе **Smalltalk** определяется следующим образом

```
Object subclass: #Dos
  instanceVariableNames: 'registers '
  classVariableNames: ''
  poolDictionaries: ''
```

Всякий раз, когда надо работать с регистрами и прерываниями, создается новый экземпляр класса **Dos** с переменной экземпляра **registers** — массивом с 18-тью элементами, каждый из которых — экземпляр класса **Integer**. В массиве **registers** отведено по два элемента для эмуляции работы со следующими регистрами процессора Intel 8086, пронумерованными таким образом, что

Регистр n	AX 0	BX 1	CX 2	DX 3	SI 4	DI 5	DS 6	ES 7
Регистр индекс	AH 1	AL 2	BH 3	BL 4	CH 5	CL 6	DH 7	DL 8

То есть, в массиве **registers** значение, хранящееся в старшем байте регистра с номером  $n$ , будет храниться по индексу  $2n + 1$ , а значение, хранящееся в младшем байте этого регистра — по индексу



$2n + 2$ . Вся работа по размещению в переменной `registers` необходимых значений выполняют следующие методы экземпляра

`setReg: regInteger to: anObject`

"Записать в регистр с номером `regInteger` значение `anObject`."

`(anObject isKindOf: Integer)`

`ifTrue: [ anObject < 0`

`ifTrue: [^self error: 'negative number not allowed']`

`ifFalse: [ registers at: regInteger * 2 + 1`

`put: ((anObject quo: 256) rem: 256);`

`at: regInteger * 2 + 2`

`put: (anObject rem: 256)]]`

`ifFalse: [registers at: regInteger * 2 + 1 put: anObject]`

`setRegHigh: regInteger to: valInteger`

"Записать в старший байт регистра с номером `regInteger` значение `anObject`."

`valInteger < 0`

`ifTrue: [^self error: 'value must be positive integer'].`

`registers`

`at: regInteger * 2 + 1 put: (valInteger rem: 256)`

`setRegLow: regInteger to: valInteger`

"Записать в младший байт регистра с номером `regInteger` значение `anObject`."

`valInteger < 0`

`ifTrue: [^self error: 'value must be positive integer'].`

`((registers at: regInteger * 2 + 1) isMemberOf: SmallInteger)`

`ifFalse: [ "Записать в старший байт регистра значение 0"`

`registers at: regInteger * 2 + 1 put: 0].`

`registers at: regInteger * 2 + 2 put: (valInteger rem: 256)`

Получить массив `registers` позволяет метод экземпляра

`registers`

`^ registers`

Выполнить прерывание с номером `anInteger` позволяет метод экземпляра

```
interrupt: anInteger
  self dosPrimitive: 0
    registers: registers
    value: anInteger
```

Кроме работы с прерываниями класс `Dos` обеспечивает и работу напрямую с аппаратурой через порты ввода/вывода.

`Dos` instance protocol

---

`inByteFromPort: portAddress` — возвращает байт, прочитанный из порта с номером `portAddress`.

`inByteFrom: portAddress` — полностью аналогично предыдущему сообщению.

`outByte: byteValue toPort: portAddress` — записывает значение `byteValue` в порт с номером `portAddress`.

---

В этих, как и во многих других методах, реальную работу с регистрами и прерываниями, выполняет примитивный метод

```
dosPrimitive: function
  registers: anArray
    value: anInteger
    "Частный — — — Выполнить DOS-функцию с номером прерывания anInteger
    и массивом значений в регистрах из registers.
    Функции пронумерованы следующим образом:
    interrupt = 0, inWord = 1, inByte = 2, outWord = 3
    outByte = 4, peek = 5, poke = 6, blockMove = 7, farCall = 8 "
    <primitive: 77>
```

Заметим, что в классе `Dos` нет методов-оболочек для целого ряда функций, реализуемых этим примитивом, поскольку в среде `Small-talk` они не безопасны.

Для демонстрации того, как используются в языке Smalltalk работа с прерываниями с помощью экземпляров класса `Dos`, приведем метод экземпляра из класса `Dos`, который определяет тип монитора компьютера, и хорошо известный из других языков программирования пример по подсчету свободного места на диске. В системе Smalltalk — это метод экземпляра из класса `Directory`.

Сначала метод из класса `Dos`:

```
monitorIsAnalog
self setRegHigh: 0 to: 16r1A;
  setRegLow: 0 to: 16r0;
  interrupt: 16r10.
(registers at: 2) = 16r1A ifFalse: [^ false].
^(registers at: 4) = 8
```

Теперь метод из класса `Directory`:

```
freeDiskSpace
"Возвращает в байтах размер свободного пространства
на диске, содержащем текущий каталог."
| aDos space regs |
aDos := Dos new.
aDos setReg: 3 to: (drive asciiValue - $a asciiValue + 1);
  setRegHigh: 0 to: 16r36; interrupt: 16r21.
regs := aDos registers.
^((regs at: 5) * 256 + (regs at: 6)) "байт на сектор"
  * ((regs at: 1) * 256 + (regs at: 2)) "байт на кластер"
  * ((regs at: 3) * 256 + (regs at: 4)) "итог!"
```

Для того, чтобы понять почему для решения поставленной задачи в регистры надо помещать то или иное число и выполнять ту или иную функцию и прерывание надо обратиться к руководству по системе MS/DOS (например, [10]), в которой описывается вызов функций операционной системы.

# 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
- [8] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Класс Collection и его подклассы. Часть 1, 2*. — Ростов-на-Дону: УПЛ РГУ, 1996
- [9] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Класс Magnitude и его подклассы*. — Ростов-на-Дону: УПЛ РГУ, 1996
- [10] Чижов А.А., *Системные программные средства ПЭВМ: Справочник* — М.: «Финансы и статистика», СП Параграф, 1990

# Содержание

<b>1</b>	<b>Протокол класса Stream</b>	<b>3</b>
1.1	Класс Stream . . . . .	4
1.2	Класс ReadStream . . . . .	11
1.3	Класс WriteStream . . . . .	12
1.4	Класс ReadWriteStream . . . . .	16
<b>2</b>	<b>Потоки генерируемых элементов</b>	<b>16</b>
<b>3</b>	<b>Класс FileStream</b>	<b>18</b>
<b>4</b>	<b>Класс Directory</b>	<b>24</b>
<b>5</b>	<b>Классы File и FileHandle</b>	<b>27</b>
<b>6</b>	<b>Класс TerminalStream</b>	<b>29</b>
<b>7</b>	<b>Класс Dos</b>	<b>32</b>
	<b>Литература</b>	<b>36</b>