Карпов В.Э.

Объектно-ориентированное программирование

Смолток. Лекция 3

Циклические конструкции

Циклы – тоже почти все «самодельные»

<число> **timesRepeat**: [блок сообщений]	Повторить блок заданное <число> раз
[блок условия] **whileFalse**: [блок сообщений]	Пока условие ложно, выполняются сообщения
[блок условия] **whileTrue**: [блок сообщений]	Пока условие истинно, выполняются сообщения
<число1> **to**:<число2> **by**:<шаг> **do**: [:<переменная> | блок сообщений]	Выполнить блок сообщений, пока значение <переменной>, изменяющее свое значение с заданным шагом, принадлежит промежутку (число1,число2)
<объект> **выполнить**: [:<переменная> | блок сообщений]	Значение <переменной> присваивается последовательно элементам <объект>
<объект> **выбрать**: [:<переменная> | <условие>]	Изменяет <объект>, удаляя элементы, не удовлетворяющие условию
<объект> **исключить**: [:<переменная> | <условие>]	Изменяет <объект>, удаляя элементы, удовлетворяющие условию
<объект> **собрать**: [:<переменная> | сообщение]	Заменяет каждый элемент <объекта> на результат выполненного сообщения

Цикл While

whileTrue (покаИстина), whileFalse (покаЛожь)

***whileTrue**: block*

"Выполняет заданный блок до тех пор, пока приемник не будет иметь значение 'ложь'. Выдает нуль"
self value ifTrue: [block value. self whileTrue: block].
^nil!

***whileFalse**: block*

"Выполняет заданный блок до тех пор, пока приемник не будет иметь значение 'истина'. Выдает нуль"
self value ifFalse: [block value. self whileFalse: block].
^nil!

Пример программы, копирующей файл ‘SOURCE.TXT’ в файл ‘DEST.TXT’:

| ввод вывод |
ввод:= Файл полноеИмя: 'SOURCE.TXT'.
вывод:=Файл полноеИмя: 'DEST.TXT'.
[ввод вКонце]
whileFalse: [вывод поместитьСледующий: ввод следующий].
ввод закрыть.
вывод закрыть.

Простой цикл

```
timesRepeat: block
    | n |
    n := self.
    [n > 0] whileTrue: [n := n - 1. block value]
```

Пример:

```
3 timesRepeat: [ Черепашка переместитьНа:
    100; повернутьНа: 120 ]
```

Арифметический цикл

***to**:limit **by**:step **do**:block*

self<=limit

ifTrue:

[block value: self.

*(self+step) **to**: limit **by**:step **do**: block]*

Пример:

sum := 0.

1 to:100 by:2 do: [:each | sum := sum + each].

Создание новых объектов

Чтобы создать объект, классу посылается сообщение о создании нового экземпляра.

Класс создает экземпляр, инициализирует и выдает созданный экземпляр в качестве ответа на сообщение.

Пример 1:

x := Array new: 10.

Для того чтобы обратиться к нужному методу экземпляра класса, необходимо, чтобы уже существовал экземпляр класса.

Пример 2:

ДемоКласс запустить.

- компилятор выдает ошибку, т.к. здесь сообщение посылается классу, а в протоколе сообщений класса такого метода нет. Метод *'запустить'* находится в протоколе сообщений экземпляра класса. Поэтому необходимо сначала создать экземпляр класса:

ДемоКласс новыйЭкземпляр запустить.

Пример 3:

```
|c|
c:= BouncingBallView  new.
c show.

BouncingBallView show
```

ПРОГРАММИРОВАНИЕ

Этапы разработки программы

1. Разработка *спецификации метода* (**предварительная постановка задачи**):
- Что будет являться объектом - адресатом?
- Что будет являться объектом - параметром?
- Что будет являться результатом?

2. Создается новый класс объектов, необходимых для решения задачи некоторой прикладной области **(анализ предметной области**).
- определяется место класса в иерархии прикладных классов (определяется его суперкласс и возможные подклассы).

3. Для каждого класса определяется протокол сообщений и внутренняя структура экземпляров.

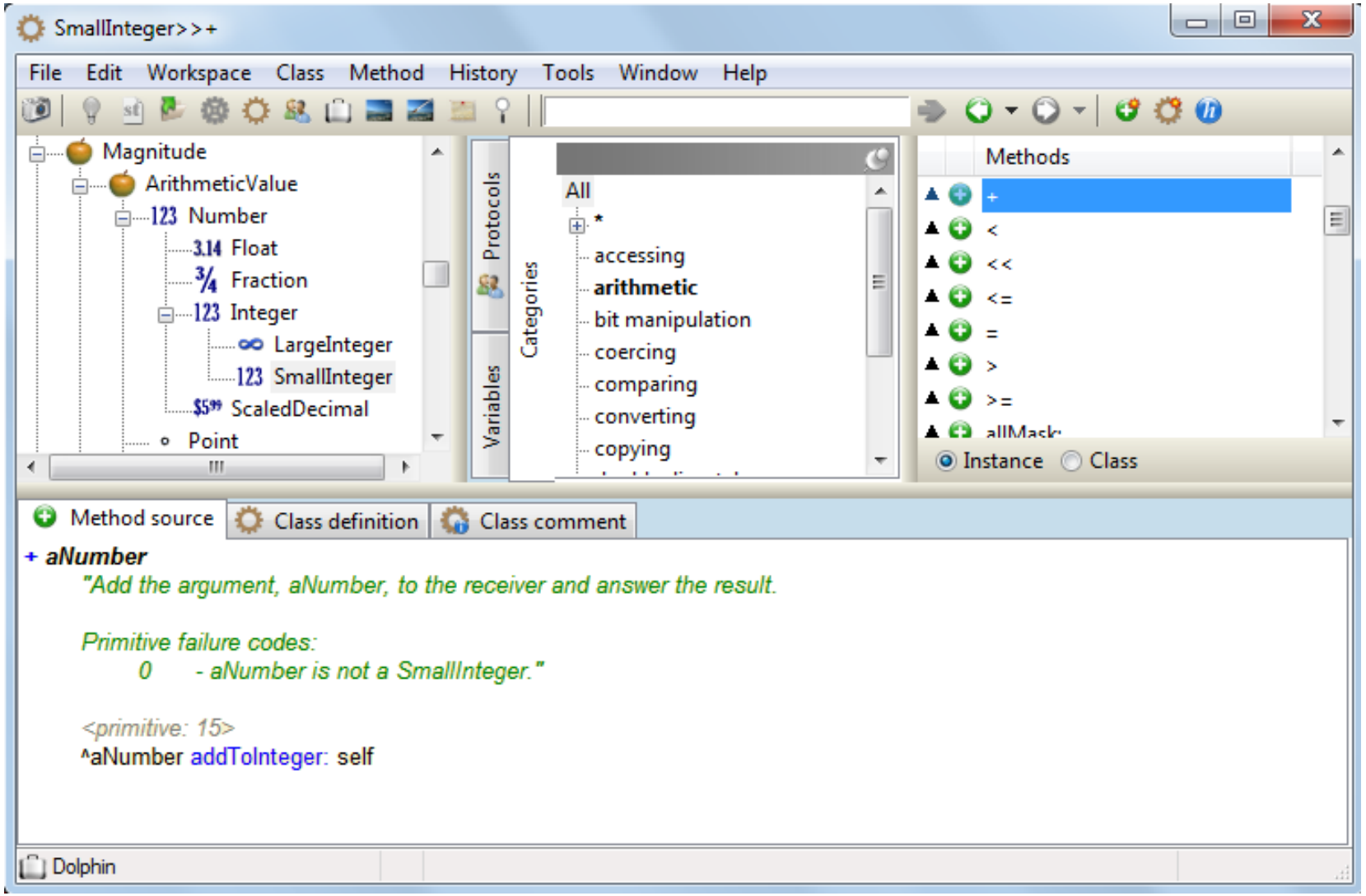
4. Алгоритм решения задачи записывается на языке взаимодействия объектов (**проектирование прикладной системы**):
- объекты порождаются,
- инициализируются,
- обмениваются сообщениями с другими объектами в целях получения результата решения задачи.

5. Новые классы реализуются с помощью имеющихся классов системы Смолток (**программированию задачи**).

Системный генератор путей

- Категории - группы классов, выделенные для некоторых областей применения.
- Классы - классы системы Смолток.
- Категории методов - группы методов классов или методов экземпляров, выделенные для удобства по признаку применения или по иному свойству.
- Методы - методы системы Смолток.

Системный генератор путей в DolphinSmalltalk



Рекурсия

```
-- Фибоначчи
-- выдать N число Фибоноччи, где N - объект-адресат
-- Числа Фибоначчи задаются следующей последовательностью:
-- x1=1, x2=1, xi=xi-1+xi-2
fibo
    self<3 ifTrue: [ ^1 ]
          ifFalse: [ ^ (self - 1) fibo + (self - 2) fibo ]
```

Пример:

```
#(1 2 3 4 5 6 7 10 20) собрать: [ :m| m fibo ]
#(1 2 3 4 5 6 7 10 20) collect: [:each | each fibo ].
```

```
#(1 1 2 3 5 8 13 55 6765)
```

Ханойские башни

1 Этап. Спецификация метода

- объект-адресат - это целое число (количество колец). $\Rightarrow$ метод будет принадлежать классу целых чисел.
- объекты-параметры - символы '1', '2', '3' (информация: с какого штырька на какой нужно переложить кольцо и какой использовать как дополнительный)
- результат - вывод на экран информации о последовательности выполнения данной задачи.

Вызов метода будет выглядеть так:

10 ханойС: '1' на: '2' через: '3'

Алгоритм

2 Этап. Реализация метода *ханойС: x на: y через: z.*

Рекурсивное решение.

- Базис. Если число колес равно 1, то нужно просто переложить это кольцо с x на y.
- Если число колес равно K, то нужно переложить со штырька x на штырек z K-1 колец, а затем переложить самое большое K-е кольцо на y и переложить все остальные K-1 колец на y.

Реализация метода

```
ханойС: x на: y через: z
"головоломка 'Ханойские башни'"
    self := 1
    ifTrue: [ СистемнаяИнформация          поместитьВсеПоследующие:
            'Переложить со штырька ',x,' на штырек ',y;
            символВК. ]
    ifFalse: [ (self - 1) ханойС: x на: z через: y.
            СистемнаяИнформация поместитьВсеПоследующие:
            'Переложить со штырька ',x,' на штырек ',y;
            символВК.
    (self - 1) ханойС: z на: y через: x ]
```

Пример использования:

```
10 ханойС: '1' на: '2' через: '3'
```

Пример метода в Smalltalk Express

```
hanoiFrom: x to: y by: z result: aStream
"головоломка 'Ханойские башни'"
self == 1
    ifTrue: [ 'Переложить со штырька ' printOn: aStream.
        x printOn: aStream.
        'на штырек ' printOn: aStream.
        y printOn: aStream.
        (10 asCharacter) printOn: aStream.]
    ifFalse: [ (self - 1) hanoiFrom: x to: z by: y result: aStream.
        'Переложить со штырька ' printOn: aStream.
        x printOn: aStream.
        'на штырек ' printOn: aStream.
        y printOn: aStream.
        (10 asCharacter) printOn: aStream.
        (self - 1) hanoiFrom: z to: y by: x result: aStream.]
```

Пример использования

```
|aStream|
aStream := WriteStream on: String new.
10 hanoiFrom: 1 to: 2 by: 3 result: aStream.
aStream contents.
```

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Три механизма:

1. параллелизм (parallelism)
2. планирование (scheduling)
3. синхронизация (synchronisation).

Три класса:

Process, *ProcessScheduler, Semaphore*

Параллелизм

Класс *Process* реализует механизм распараллеливания

[f computeFunc] ***fork***

- *newProcess* создать новый процесс, но не выполнять его;
- *resume* выполнить остановленный процесс;
- *suspend* остановить процесс;
- *terminate* закончить процесс.

Например:

FProcess := [f computeFunc] newProcess,

сформируется новый процесс, однако он будет находиться в состоянии ожидания и выполняться не будет. Выполнение его начнется только после того, как будет выполнено сообщение:

FProcess resume.

Диспетчеризация. Класс *ProcessScheduler*

ProcessScheduler имеет единственный экземпляр, указателем на который является глобальная переменная *Processor*

```
Processor fork: block1
```

Алгоритм диспетчеризации реализуется через простую систему приоритетов (priority system).

Приоритеты процессов в Smalltalk Express определяются методами класса ProcessScheduler:
userPriority, highUserPriority, lowUserPriority, topPriority

В SmallTalk Express:

```
userPriority
        ^4!
highUserPriority
        ^6!
lowUserPriority
        ^2!
topPriority
        ^7!
```

yield – остановить процесс с учетом приоритета

```
[Float computeFunc] forkAt:Processor userBackgroundPriority
```

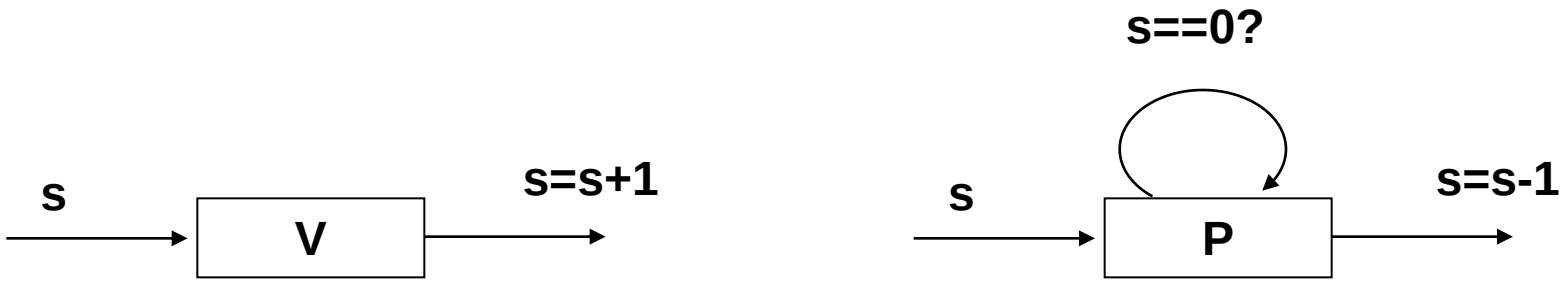
Пример работы (Smalltalk Express)

```
1.  |aStream n block1 block2 w nfibo flimit|
2.  w := TextWindow windowLabeled: 'Fibo' frame: (100@100 extent: 400@200).
3.  flimit := 25.
4.  aStream := WriteStream on: String new.
5.  block1 := [ n := 0.
6.  [n<flimit] whileTrue:
7.                  [n:=n+1.
8.                  nfibo := ( n fibo).
9.                  nfibo printOn: aStream. ' ' printOn: aStream.
10.                 w nextPutAll: ' P-1 '.
11.                 w nextPutAll: (n radix:10).
12.                 w nextPutAll: ' : '.
13.                 w nextPutAll: (nfibo radix:10); cr.].
14.                 w nextPutAll: 'Process 1 done.'; cr.].
15. block2 := [ n := 0.
16. [n<flimit] whileTrue:
17.                 [n:=n+1.
18.                 nfibo := ( n fibo).
19.                 nfibo printOn: aStream. ' ' printOn: aStream.
20.                 w nextPutAll: ' P-2 '.
21.         w nextPutAll: (n radix:10).
22.         w nextPutAll: ' : '.
23.         w nextPutAll: (nfibo radix:10); cr.].
24.         w nextPutAll: 'Process 2 done.'; cr.].
25. block1 forkAt: (ProcessScheduler new lowUserPriority).
26. block2 forkAt: (ProcessScheduler new lowUserPriority).
```

Результаты работы отображаются как в окне, так и в потоке *aStream (aStream contents).*

Синхронизация. Класс *Semaphore*

- Семафоры Дейкстры
- Функции P и V



Задача о кольцевом буфере

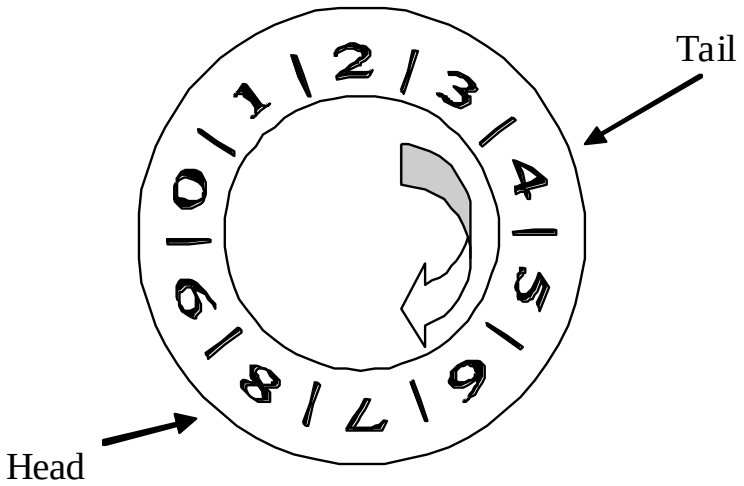
Буфер на N ячеек

Голова (Head) и хвост (Tail)

Head – первая свободная ячейка,

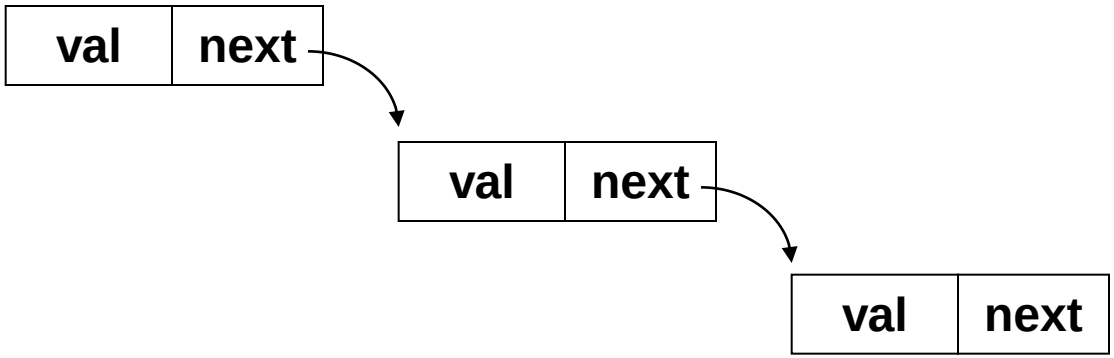
Tail – последняя занятая.

Пишем в голову, читаем из хвоста.



Класс «Ячейка»

```
" Класс BUFFER "
Object subclass: #Buffer
instanceVariableNames:
'next val '
classVariableNames: "
poolDictionaries: " !
!Buffer methods !
next
    ^next.!
next: link
    next := link.!
put: data
    val := data.!
val
    ^val.! !
```



Класс «Кольцевой буфер»

```
Object subclass: #RingBuffer
instanceVariableNames:
'empty full head tail fetchprotect storeprotect '
-- семафоры:
-- empty - количество данных в буфере
-- full - количество свободных мест
-- head, tail - указатели на начало и конец
-- fetchprotect и storeprotect - вспомогательные переменные, обеспечивающие
-- взаимоисключение
classVariableNames: ''
poolDictionaries: '' ! !
!RRingBuffer class methods !
create: size
    ^self new init: size.! !
!RRingBuffer methods !
init: size
    head := tail := Buffer new.
    1 to: size - 1 do: [ :i | tail next: Buffer new. tail := tail next].
    tail next: head.
    tail := head.
    empty := Semaphore new.
    full := Semaphore new.
    1 to: size do: [:i | full signal].
    fetchprotect := Semaphore new signal.
    storeprotect := Semaphore new signal.!
```

Методы fetch и store

```
fetch
    |data|
    fetchprotect wait.
    empty wait.
    data := tail val.
    tail := tail next.
    full signal.
    fetchprotect signal.
    ^data.!
store: data
    storeprotect wait.
    full wait.
    head put: data.
    head := head next.
    empty signal.
     storeprotect signal.! !
```

Пример использования

```
1.  |n block1 block2 w nfibo rb d flimit|
2.  flimit := 25.
3.  rb := RingBuffer new init: 20.
4.  -- или так: rb := RingBuffer create: 20.
5.  w := TextWindow windowLabeled: 'Ring Buffer' frame: (100@100 extent: 400@200).
6.  block1 := [ n := 0.
7.                  [n<flimit] whileTrue:
8.                          [n:=n+1.
9.                          nfibo := ( n fibo).
10.                         rb store: nfibo.
11.                         ].
12.         w nextPutAll: 'Process 1 done.'; cr.].
13. block2 := [ n := 0.
14.                 [n<flimit] whileTrue:
15.                         [n := n+1.
16.                         d := rb fetch.
17.                         w nextPutAll: (d radix:10); cr.
18.                         ].
19.         w nextPutAll: 'Process 2 done.'; cr.].
20. block1 forkAt: Processor lowUserPriority. -- или block1 forkAt: 1.
21. block2 forkAt: Processor lowUserPriority. -- или block1 forkAt: 1.
```