

Ю.А. Кирютенко, В.А. Савельев

Объектно-ориентированное программирование
и язык Smalltalk.

Класс Magnitude и его подклассы.

Ростов-на-Дону

1997

Объектно-ориентированное программирование и язык Smalltalk.

Класс **Magnitude** и его подклассы.

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**, посвященных общим концепциям и синтаксису языка, интерфейсу пользователя и среде программирования, классу **Collection** и его подклассам. В данной работе описывается протокол и реализация системного класса **Magnitude** и его 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 № 5 от 10 декабря 1996 года.

Настоящие методические указания набраны в системе \LaTeX с использованием кириллических шрифтов семейства ЛН (дизайнеры О. Лапко и А. Ходулев).

1 Класс Magnitude

Является ли одно число больше, чем другое? Следует ли эта дата за той? Стоит ли некоторая буква перед или за указанной буквой в алфавите? Общий протокол для ответа на все эти вопросы предоставляет абстрактный класс **Magnitude** (Величина). В нем обеспечиваются любые сравнения линейно упорядоченных объектов. Как суперкласс класс **Magnitude** включает подклассы **Time** (Время), **Date** (Дата), **Number** (Число), **Character** (Символ). Полная иерархия класса **Magnitude** в языке **Smalltalk** следующая

```
Magnitude
  Association
  Character
  Date
  Time
  Number
    Float
    Fraction
    Integer
      LargeNegativeInteger
      LargePositiveInteger
      SmallInteger
```

Magnitude instance protocol

< aMagnitude — указывает, меньше ли приемник сообщения аргумента **aMagnitude**

<= aMagnitude — указывает, меньше или равен приемник сообщения аргумента **aMagnitude**

> aMagnitude — указывает, больше ли приемник сообщения аргумента **aMagnitude**

>= aMagnitude — указывает, больше или равен приемник сообщения аргумента **aMagnitude**

between: min and: max — указывает, будет ли приемник сообщения больше или равен аргумента **min**, и меньше или равен аргумента **max**.

min: aMagnitude — из двух величин, приемника сообщения и аргумента, возвращает ту, которая меньше.

max: aMagnitude — из двух величин, приемника сообщения и аргумента, возвращает ту, которая больше.

Отметим, что метод, ассоциированный с сообщением `=`, реализован как `self implementedBySubclass`. Поэтому каждый подкласс класса `Magnitude` обязательно переопределяет это сообщение. Вместе с тем, протокол сравнения на совпадение `==`, `~=`, `~~` наследуется из класса `Object`. Используя целые числа как один из видов объектов класса `Magnitude`, получим

выражение	результат
-----------	-----------

<code>3 <= 4</code>	<code>true</code>
<code>3 > 4</code>	<code>false</code>
<code>5 between: 2 and: 6</code>	<code>true</code>
<code>34 max: 45</code>	<code>45</code>

Стоит заметить, что во многих методах, определяемых подклассами класса `Magnitude` и выполняющих фактическую работу с величинами, широко используются примитивные методы системы. Исключение составляет только классы `Association` и `Fraction`.

2 Ассоциативные пары

Пожалуй наиболее простым подклассом класса `Magnitude` является класс `Association` (АссоциативнаяПара), экземпляры которого играют очень важную роль в классе `Dictionary` и его подклассах.

```
Magnitude subclass: #Association
  instanceVariableNames: ' key value '
  classVariableNames: ' '
  poolDictionaries: ' '
```

Как видно из определения класса, каждый экземпляр класса **Association** имеет две переменные **key** (ключ) и **value** (значение), первая из которых обязательно должна ссылаться на объект, в интерфейс которого входят сообщения сравнения **<**, **<=**, **>**, **>=**, **=**. Ключами ассоциативных пар могут быть, например, числа, строки или символы. Сами ассоциативные пары включают в свой интерфейс вышеперечисленные сообщения сравнения, выполняя их над ключами. Второй аргумент — **value** — может ссылаться на любой объект системы. Создать экземпляр класса **Association** позволяют следующие сообщения, входящие в его протокол

Association class protocol

key: aKey — возвращает экземпляр класса **Association** с ключом **aKey**.

key: aKey value: anObject — возвращает экземпляр класса **Association** с ключом **aKey** и значением **anObject**.

Помимо сообщений сравнения, протокол сообщений, посылаемых экземплярам этого класса содержит унарные сообщения **key** и **value** для доступа к переменным экземпляра, ключевые сообщения **key:** и **value:** для изменения соответствующих переменных. Кроме того, здесь переопределяются сообщения **printOn:** и **storeOn:**, наследуемые из класса **Object**.

Реализация сообщений приведенного выше протокола класса достаточно проста. В обоих случаях сначала с помощью сообщения **self new** создается новый экземпляр класса **Association** с переменными **key** и **value**, ссылающимися на **nil**, а затем созданному экземпляру посылаются ключевые сообщения экземпляра, инициализирующие эти переменные.

Association class methods

key: anObject

 "Возвращает экземпляр класса **Association** с заданным ключом"
 [^]**self new key: anObject**

key: aKey value: anObject

"Возвращает экземпляр класса Association
с заданными ключом и значением"

^(self key: aKey) value: anObject

Association methods

key: anObject

"Задает значение ключа (первого объекта) приемника.
Возвращает приемник в качестве результата"

key := anObject

value: anObject

"Задает значение (второй объект) приемника.
Возвращает приемник в качестве результата."

value := anObject

Обратите внимание, что в классе `Association` существует и сообщение класса и сообщение экземпляра с одним и тем же селектором сообщения `key:`. Никакой путаницы при этом не возникает. Когда сообщение посылается классу как объекту системы, поиск соответствующего метода начинается в метаклассе класса `Association` и там находится. Когда же сообщение посылается экземпляру класса `Association`, поиск нужного метода начинается в самом классе.

3 Класс Character

Класс `Character` представляет особый интерес, поскольку представляет неизменяемые объекты системы. Он включен как подкласс в класс `Magnitude` потому, что все экземпляры класса `Character` образуют линейно упорядоченное множество, то есть для двух символов из этого множества всегда можно сказать, какой из символов предшествует (`<`) или следует (`>`) за другим. Всего в системе существует 256 экземпляров класса `Character`, каждый из которых ассоциируется с кодом ASCII из расширенной ASCII-таблицы символов и определяется переменной экземпляра `asciiInteger`. Класс `Character` имеет доступ к словарию `CharacterConstants` (СимвольныеКонстанты), который содержит такие символы как, например, `Tab` (Табуляция),

Space (Пробел), Cr (ВозвратКаретки) и другие. Это следует из приводимого ниже определения класса

```
Magnitude subclass: #Character
  instanceVariableNames: ' asciiInteger '
  classVariableNames: ' '
  poolDictionaries: ' CharacterConstants '
```

Протокол доступа к всегда существующим в системе экземплярам класса **Character** приводится ниже. Кроме того, в системе каждый экземпляр класса **Character** может быть представлен еще и литерально, то есть записан в виде символа алфавита, которому предшествует знак \$ (доллара). Например, \$A представляет прописную английскую букву 'A'.

Character class protocol

value: anInteger — возвращает экземпляр класса **Character**, ASCII код которого равен аргументу **anInteger**. Например, выражение **Character value: 65** возвращает прописную английскую букву 'A'.

digitValue: anInteger — возвращает экземпляр класса **Character**, чье числовое значение есть аргумент **anInteger**. Например, возвращает \$9, если аргумент есть 9, возвращает \$0, если аргумент есть 0, возвращает \$A, если аргумент есть 10 и \$Z, если аргумент есть 35. Обычно, используются только символы до \$F (при 16-ричной системе счисления).

Сообщения к экземплярам класса **Character** поддерживают доступ к их ASCII-значениям и к цифровым значениям этих экземпляров, производят проверку типа символа. Единственное назначение экземпляра класса **Character** в системе — хранить свое значение, которое не может быть изменено. Объекты системы, которые не могут изменить своего внутреннего статуса, называются неизменяемыми объектами. Кроме того, все 256 экземпляров этого класса создаются во время инициализации системы и не изменяются. Как

только требуется новый символ с ASCII-кодом от 0 до 255, устанавливается ссылка на уже существующий символ.

Character instance protocol

asciiValue — возвращает число, соответствующее ASCII-коду приемника.

asSymbol — преобразует символ-приемник, экземпляр класса **Character**, в экземпляр класса **Symbol**.

digitValue — возвращает число, соответствующее цифровому представлению приемника сообщения (см. сообщение **digitValue:**).

isAlphaNumeric — возвращает **true**, если приемник буква или цифра.

isDigit — возвращает **true**, если приемник цифра.

isLetter — возвращает **true**, если приемник буква.

isLowercase — возвращает **true**, если приемник строчная буква.

isUppercase — возвращает **true**, если приемник прописная буква.

isSeparator — возвращает **true**, если приемник является одним из символов-разделителей: **cr**, **line-feed**, **form-feed**, **space**, **tab**.

isVowel — возвращает **true**, если приемник одна из гласных букв английского алфавита.

Кроме того, существует возможность преобразовывать символы из прописных в строчные и наоборот (**asUppercase**, **asLowercase**).

Поддерживается так же протокол сравнения экземпляров класса **Character** в алфавитном порядке. Например, мы желаем узнать предшествует ли одна строка другой в телефонной книге. Элементы строки — символы, и строка (экземпляр класса **String**) понимает сообщение **at: anIndex**, которое возвращает символ из строки с индексом **anIndex**; таким образом, **'abcde'** **at: 2** есть **\$b**. Определим специальный метод в классе **String**, с селектором **min:**, который возвращает ту из двух строк, приемника сообщения или его аргумента, что стоит раньше согласно упорядочивания по алфавиту.


```

min: aString
  1 to self size do:
    [:index|
      (index > aString size) ifTrue: [^ aString].
      (self at: index) > (aString at: index) ifTrue: [^ aString].
      (self at: index) < (aString at: index) ifTrue: [^ self].
    ]
  ^ self

```

Алгоритм этого метода состоит из следующих очевидных действий. Число итераций ограничивается числом символов в приемнике сообщения. Итерации прекращаются в следующих нижеперечисленных ситуациях

- 1) Все предыдущие символы совпали, но в аргументе `aString` больше нет символов (`index > aString size`).
- 2) Следующий символ в приемнике сообщения расположен после соответствующего символа в аргументе `aString` (`(self at: index) > (aString at: index)`).
- 3) Следующий символ в приемнике сообщения расположен до соответствующего символа в аргументе `aString`.

Рассмотрим примеры. Пусть сравниваются строки `'abcd'` и `'abc'`, причем первая — приемник сообщения, тогда при значении `index = 4`, будет возвращена строка `'abc'` (реализовалась ситуация (1)). Если же строка `'abde'` — приемник сообщения - сравнивается со строкой `'abcd'`, то когда `index = 3`, условие `d > c` выполняется и метод возвращает строку `'abcd'`. В том случае, когда сравниваются строки `'az'` и `'by'`, причем первая — приемник сообщения, реализуется ситуация (3) и метод возвратит строку `'az'`. В том случае, когда приемник сообщения имеет меньше символов, чем аргумент, и даже тогда, когда приемник составляет начальную подстроку аргумента, условие `index > aString size` никогда не реализуется, и либо возникнут ситуации (2), (3), либо ни одна из них не реализуется и тогда в методе выполнится выражение `^ self`, которое вернет приемник сообщения как результат. Последнее произойдет, например, при сравнении строк `'abc'` и `'abcd'`, когда первая — приемник сообщения.

Отметим, что арифметические операции с символами не поддерживаются. Попытка их выполнить будет приводить к сообщению об ошибке.

4 Класс Date

После того, как определен общий протокол класса **Magnitude**, есть возможность дополнить его новым протоколом, который бы и поддерживал арифметические операции и выполнял специальные линейные измерения. Одним из таких расширений является класс **Date**:

```
Magnitude subclass: #Date
  instanceVariableNames: ' day '
  classVariableNames: ' MonthNames MonthStrings '
  poolDictionaries: ' '
```

Экземпляр класса **Date** (**Дата**) представляет собой конкретный день (**day**) от начала юлианского календаря. Этот день существует в конкретном месяце и году. Класс **Date** имеет две переменные класса — словари с именами **MonthNames** и **MonthStrings**, состоящие из ассоциативных пар, в которых значения — индекс месяца в году, а ключи — имена месяцев, выраженные в первом словаре экземплярами класса **Symbol**, а во втором — экземплярами класса **String**. Например, в первый словарь входят ассоциативные пары вида (**#Jan 1**) и (**#January 1**), а во второй — (**'jan' 1**) и (**'january' 1**). Классу **Date** известна некоторая очевидная информация:

- в неделе 7 дней, каждый день имеет имя и соответствующий индекс 1,...,7;
- в году 12 месяцев, каждый месяц имеет имя и соответствующий индекс 1,...,12;
- в каждом месяце может быть 28, 29, 30 или 31 день;
- каждый год может быть обычным в 365 дней или високосным в 366 дней.

Для первого знакомства с протоколом класса **Date** рассмотрим сообщения, поддерживающие общие запросы к классу **Date**.

dayOfWeek: dayName — возвращает для имени дня недели, заданного аргументом **dayName**, его индекс (1,2,...или 7).

nameOfDay: dayIndex — возвращает имя дня недели по его индексу, где 1 – Monday, 2 – Tuesday, и т.д., до 7 – Sunday.

indexOfMonth: monthName — возвращает для имени месяца, заданного аргументом **monthName**, его индекс (1,2,...,12).

nameOfMonth: monthIndex — возвращает имя месяца по его индексу, где 1 – January, 2 – February, и т.д..

daysInMonth: monthName forYear: yearInteger — возвращает число дней в месяце с именем **monthName** в году **yearInteger** (год должен быть известен, чтобы определить является он високосным или нет).

daysInYear: yearInteger — возвращает число дней в году **yearInteger**.

leapYear: yearInteger — возвращает 1, если год **yearInteger** — високосный, иначе возвращает 0.

dateAndTimeNow — возвращает массив, первый элемент которого есть текущая дата (экземпляр класса **Date**), а второй элемент — текущее время (экземпляр класса **Time**).

Таким образом, мы можем послать классу **Date** такие сообщения

выражение

результат

Date daysInYear: 1982

365

Date dayOfWeek: #Wednesday

3

Date nameOfMonth: 10

October

Date leapYear: 1972

1 (високосный)

Date daysInMonth: #February forYear: 1972

29

В классе **Date** обычно для названий дней недели и месяцев используются общепринятые (в англоязычных странах) сокращения.

Для создания экземпляров класса **Date** могут использоваться следующие четыре сообщения. Одно из них, а именно **today**, особенно часто используется при установке даты создания файла.

Date class protocol

today — возвращает дату, когда было послано это сообщение.

fromDays: dayCount — возвращает дату, которая соответствует дню **dayCount** до или после 1-го января 1901 г (в зависимости от знака аргумента **dayCount**).

newDay: day month: monthName year: yearInteger — возвращает дату, которая соответствует дню **day** в указанном месяце **monthName** в году **yearInteger**.

newDay: dayCount year: yearInteger — возвращает дату, которая соответствует дню **dayCount** с начала года **yearInteger**.

Вот примеры создания экземпляров класса **Date**, использующие эти сообщения.

выражение	результат
<code>Date today</code>	3 October 1996
<code>Date fromDays: 200</code>	20 July 1901
<code>Date newDays: 6 month: #Feb year: 96</code>	6 February 1996
<code>Date newDays: 3 year: 96</code>	3 January 1996

Протоколом класса **Date** поддерживаются следующие простые арифметические операции с датами.

addDays: dayCount — возвращает дату через заданное число дней **dayCount** после даты-приемника сообщения.

subtractDays: dayCount — возвращает дату через заданное число дней **dayCount** до даты-приемника сообщения.

subtractDays: aDate — возвращает целое число, представляющее число дней между приемником сообщения и аргументом **aDate**.

Такая арифметика с датами полезна. Например, надо вычислить дату возвращения книги в библиотеку или число дней просрочки возврата. Если **dueDate** — это дата возвращения книги, то число дней просрочки мы получим, вычисляя выражение

Date today subtractDate: dueDate.

Если, выданная сегодня книга, должна быть возвращена через две недели, то дата возврата книги в библиотеку вычисляется выражением

Date today addDays: 14.

5 Класс Time

Определение класса **Time** в системе **Smalltalk** следующее

```
Magnitude subclass: #Time
  instanceVariableNames: 'seconds'
  classVariableNames: 'TimeTickOn ValueArray'
  poolDictionaries: ''
```

Экземпляр класса **Time** (Время) представляет некоторую конкретную секунду в течение дня, сохраняемую в переменной экземпляра **seconds**. Считается, что день начинается в полночь. Кроме того, класс **Time** имеет две переменные класса. В переменной

ValueArray — четырехмерном массиве — храниться текущее время, состоящее из часов, минут, секунд, миллисекунд. А переменная **TimeTickOn** — логическая величина (**true** или **false**) — флаг доступа к прерыванию системного таймера.

Time class protocol

millisecondClockValue — возвращает число миллисекунд после последней установки часов на 0.

millisecondsToRun: timeBlock — возвращает число миллисекунд, необходимых для выполнения блока выражений **timeBlock**.

totalSeconds — возвращает общее число секунд, прошедшее с 1 января 1901 года, скорректированное на часовой (временной) пояс.

dateAndTimeNow — возвращает экземпляр класса **Array**, чей первый элемент — текущая дата (экземпляр класса **Date**), а второй элемент — текущее время (экземпляр класса **Time**).

now — возвращает время, в которое было послано сообщение, читая текущее время на системных часах компьютера.

fromSeconds: secondCount — возвращает время, которое представляет число секунд **secondCount** после полуночи.

Сообщение **millisecondsToRun: timeBlock** - единственный неочевидный запрос в этом протоколе. Рассмотрим пример

Time millisecondsToRun: [Date today].

В результате его выполнения мы получим число миллисекунд, которые необходимы системе, чтобы вычислить текущую дату.

Протокол доступа к экземплярам класса **Time**, обеспечивает получение информации о числе часов, минут, секунд в рассматриваемом экземпляре. Поддерживаются и арифметические операции.

addTime: timeAmount — возвращает экземпляр класса **Time**, который показывает время позже времени-приемника на время, заданное аргументом **timeAmount**.

subtractTime: timeAmount — возвращает экземпляр класса **Time**, который показывает время раньше времени-приемника на время, заданное аргументом **timeAmount**.

В этих сообщениях аргумент **timeAmount** — экземпляр либо класса **Time**, либо класса **Date**. Чтобы желаемые операции можно было проводить, система должна приводить экземпляры этих классов к общей единице измерения. Такой единицей является секунда. Для класса **Time**, перевод экземпляра класса в секунды дает число секунд после полуночи. Для класса **Date**, перевод экземпляра класса в секунды дает число секунд между временем 1-го января 1901 года и тем же временем рассматриваемой даты. Каждый из этих классов умеет правильно реагировать на сообщение **asSeconds**.

Арифметические операции с экземплярами класса **Time** аналогичны таким же операциям с экземплярами класса **Date**. Предположим, что Некто работает над проектом при условии почасовой оплаты 5,000 в час. Тогда, начиная работать во время **startTime** и работая без перерыва до настоящего момента, Некто должен получить за полностью отработанные часы сумму

$$(\text{Time now subtractTime: startTime}) \text{ hours} * 5000.$$

6 Протокол числовых классов

Система **Smalltalk** — однородная среда программирования, в которой объекты взаимодействуют посредством передачи сообщений. Утверждение, что даже сложение двух целых чисел следует рассматривать как результат послышки сообщения, на первых порах может вызвать некоторое непонимание. Но опыт показал, что преимущества предельно единообразного построения языка программирования, перевешивают некоторые неудобства при его реализации. И

теперь в арифметике потерь (по сравнению с процедурными языками программирования) почти нет, а преимущества сохранились.

Различные типы числовых величин представляются в системе различными классами, которые реализованы таким образом, что все числа ведут себя так, как если бы они принадлежали наиболее общему типу. Общий протокол для всех числовых объектов наследуется из класса **Number**, который имеет три подкласса: **Float**, **Fraction**, **Integer**. В свою очередь класс **Integer** имеет три подкласса: **SmallInteger**, **LargeNegativeInteger**, **LargePositiveInteger**. Классы **Number** и **Integer** являются абстрактными классами и только определяют разделяемый всеми протокол, но не дают индивидуальных представлений для числовых величин.

В отличие от большинства других объектов, которые могут изменять свое внутреннее состояние, единственное состояние числа — это его числовое значение, которое не может изменяться.

6.1 Протокол класса **Number**

Класс **Number** — абстрактный подкласс класса **Magnitude**, который не имеет ни переменных экземпляра, ни переменных класса, ни разделяемых с другими классами пулов. Он только определяет сообщения, поддерживающие стандартные арифметические операции и операции сравнения. Большинство из них реализуются подклассами класса **Number**, поскольку существенно зависят от представления числовых величин.

Number instance protocol

- + **aNumber** — возвращает сумму приемника и аргумента **aNumber**.
- **aNumber** — возвращает разность между приемником и аргументом **aNumber**.
- * **aNumber** — возвращает произведение приемника на аргумент **aNumber**.
- / **aNumber** — возвращает результат деления приемника на аргумент **aNumber**. Заметим, что деление вычисляется с максимальной возможной точностью, и, если точное деление невозможно, результат представляется экземпляром класса **Fraction**.

// **aNumber** — возвращает целую часть деления приемника на аргумент **aNumber**, с усечением к минус бесконечности.

\\ **aNumber** — возвращает целочисленный остаток от деления приемника на аргумент **aNumber**, с усечением нецелого остатка к минус бесконечности.

abs — возвращает абсолютную величину приемника.

negated — возвращает число, которое является противоположным к приемнику сообщения.

quo: aNumber — возвращает целую часть деления приемника на аргумент **aNumber** с усечением к нулю.

rem: aNumber — возвращает целочисленный остаток от деления приемника на аргумент **aNumber** с усечением к нулю.

reciprocal — возвращает результат деления 1 на приемник сообщения; сообщает об ошибке, если приемник равен 0.

Обратите внимание, что арифметические сообщения, которые возвращают целочисленное частное и целочисленный остаток, определены двумя способами. В одних округление происходит в сторону минус бесконечности (// и \\), а в других — в сторону числа 0 (**quo:** и **rem:**). Для положительных чисел результат всегда один и тот же, так как 0 и минус бесконечность располагаются в одном направлении от числа, а для отрицательных — разный.

выражение

результат

$1 + 10$

11

$5.6 - 3$

2.6

$5 - 10.2$

-5.2

$(-4) \text{ abs}$

4

$6 / 2$

3

$7 / 2$

(7/2)— экземпляр класса Fraction

7 reciprocal

(1/7)— экземпляр класса Fraction

$7 \text{ quo: } 2$

3

$(7 \text{ quo: } 2) + 1$

4

7 quo: 2 + 1	2 (вычисляется как 7 quo: (2 + 1))
7 rem: 2	1
7 // 2	3
7 \\ 2	1
-7 quo: 2	-3
-7 rem: 2	-1
-7 // 2	-4
-7 \\ 2	1

Дополнительно, протокол класса **Number** определяет математические функции

Number instance protocol

exp — возвращает вещественное число, которое равно экспоненте от приемника сообщения.

ln — возвращает вещественное число, которое равно натуральному логарифму от приемника сообщения.

log: aNumber — возвращает вещественное число, которое равно логарифму по основанию **aNumber** от приемника.

raisedTo: aNumber — возвращает вещественное число, равное получателю, возведенному в степень **aNumber**.

raisedToInteger: anInteger — возвращает вещественное число, равное получателю, возведенному в целую степень **anInteger**.

sqrt — возвращает вещественное число, которое равно положительному квадратному корню от приемника.

squared — возвращает вещественное число, которое равно произведению приемника на себя.

Приведем примеры, в которых используются такие сообщения.

выражение	результат
-----------	-----------

2.718282 ln	1.0
2 exp	7.38906
4 log: 2	2.0
6 log: 2	2.58496
6 raisedTo: 1.2	8.58579
6 raisedToInteger: 2	36
64 sqrt	8
8 squared	64

Чтобы определить будут ли числа четными или нечетными, положительными или отрицательными, надо воспользоваться следующими тестирующими сообщениями

Number instance protocol

even — возвращает **true**, если приемник - четное число.

odd — возвращает **true**, если приемник - нечетное число.

negative — возвращает **true**, если приемник меньше 0.

positive — возвращает **true**, если приемник больше или равен 0.

strictlyPositive — возвращает **true**, если приемник больше 0.

sign — возвращает 1, если приемник больше 0, возвращает -1, если приемник меньше нуля, иначе возвращает 0.

Свойства чисел подвергаться усечению или округлению описывается следующим протоколом.

Number instance protocol

ceiling — возвращает ближайшее целое число, не меньшее получателя.

floor — возвращает целую часть деления приемника на 1 (с усечением к минус бесконечности).

rounded — возвращает ближайшее к получателю целое число.

roundTo: aNumber — возвращает ближайшее к получателю вещественное число кратное аргументу **aNumber**.

truncated — возвращает ближайшее к получателю целое число, не превосходящее его по абсолютной величине.

truncateTo: aNumber — возвращает ближайшее к получателю вещественное число кратное аргументу **aNumber** и не превосходящее получатель по абсолютной величине.

Вот примеры выражений с такими сообщениями.

выражение	результат
-----------	-----------

16.32 ceiling	17
16.32 floor	16
-16.32 ceiling	-16
-16.32 floor	-17
-16.32 truncated	-16
16.32 truncated	16
16.32 truncateTo: 5	15
16.32 truncateTo: 5.1	15.3
16.32 rounded	16
16.32 roundTo: 6	18
16.32 roundTo: 6.3	18.9

Протокол класса **Number** включает сообщения для преобразования градусов в радианы и наоборот.

Number instance protocol

degreesToRadian — получатель представлен в градусах, возвращает его представление в радианах.

radianToDegrees — получатель представлен в радианах, возвращает его представление в градусах.

Так, например, $30 \text{ degreesToRadian} = 0.523599$, $90 \text{ degreesToRadian} = 1.570796$.

Тригонометрические и логарифмические функции включены в протокол математических функций. Приемник для тригонометрических функций `cos`, `sin`, `tan` задается в радианах. Результат, возвращаемый сообщениями `arcSin`, `arcCos`, `arcTan`, так же представляется в радианах. В следующих примерах 30 градусов — это 0.523599 радиан, 90 градусов — это 1.570796 радиан.

выражение	результат
<hr/>	
0.523599 sin	0.5
0.523599 cos	0.866025
0.523599 tan	0.57735
1.570796 sin	1.0
0.57735 arcTan	0.523599
1.0 arcSin	1.5708

Когда складываются два числа из одного класса, результат, как правило — число того же класса. Но когда, например, два операнда принадлежат классу `SmallInteger`, а их сумма велика, чтобы ее можно было представить экземпляром класса `SmallInteger`, результат будет представлен как экземпляр класса `LargerPositiveInteger` или `LargerNegativeInteger`. Определение класса результата, когда операнды принадлежат разным классам, несколько сложнее. Здесь используется два следующих критерия, во-первых, потерянной информации должно быть как можно меньше, во-вторых, должен выполняться коммутативный закон: независимо от того, какой операнд является приемником сообщения, а какой аргументом, результат должен быть один и тот же. Например, $3.1 * 4$ вернет тот же самый результат, что и $4 * 3.1$.

Представление результата при операциях над числами из разных классов, определяется численной мерой общности, приписанной каждому числовому классу. Каждый класс должен "уметь" превращать свои экземпляры в эквивалентные им экземпляры класса с большей общностью. В системе `Smalltalk` иерархия числовых классов по мере убывания их общности следующая

`Float`, `Fraction`, `LargeNegativeInteger`, `LargePositiveInteger`, `SmallInteger`.

Несколько необычна часть протокола класса `Number`, связанная с созданием экземпляра класса `Interval` (Интервал), который представляет арифметическую прогрессию.

Number instance protocol

`to: stop` — возвращает экземпляр класса `Interval` — набор чисел от числа-приемника до числа `stop`, каждый следующий элемент которого вычисляется прибавлением 1 к предыдущему.

`to: stop by: step` — аналогично предыдущему, но следующий элемент вычисляется прибавлением величины `step`.

`to: stop do: aBlock` — создает экземпляр класса `Interval` — набор чисел от приемника до числа `stop`, каждый следующий элемент которого вычисляется прибавлением 1 к предыдущему; вычисляет аргумент `aBlock` для каждого числа этого набора.

`to: stop by: step do: aBlock` — аналогично предыдущему, но следующий элемент вычисляется прибавлением величины `step`.

Итак, если вычислим выражения

```
a := 0.  
10 to: 100 by: 10 do: [:each | a := a + each]
```

то последнее вычисленное значение будет равно 550.

Если `arr := #('one' 'two' 'three' 'four' 'five')`, то в результате вычисления выражения

```
1 to: arr size do: [:index | arr at: index put: ((arr at: index) at:1)]
```

будет возвращен массив `#('o' 't' 't' 'f' 'f')`.

Хотя во всех предыдущих примерах числа записывались в десятичной системе счисления, система позволяет использовать и другие системы счисления при представлении экземпляров всех числовых классов. Для этого используется префикс основания системы счисления, предшествующий цифрам экземпляра любого числового

класса. Префикс основания системы счисления состоит из величины основания системы счисления (всегда выраженной в *ДЕСЯТИЧНОЙ* системе счисления) и следующей за ней буквы "r". Следующие примеры определяют числа в восьмеричной системе счисления с указанием соответствующего им десятичного значения.

Восьмеричная	Десятичная
8r377	255
8r153	107
8r34.1	28.125
8r-37	-31

Когда основание системы счисления больше 10, для цифр больших 9 используются заглавные латинские буквы, начиная с "A". Следующие примеры определяют числа в 16-ричной системе счисления и соответствующие им десятичные значения.

Шестнадцатеричная	Десятичная
16r106	262
16rFF	255
16rAC.DC	172.859
16r-1.C	-1.75

Числа также могут быть выражены в экспоненциальной форме с помощью экспоненциального суффикса. Экспоненциальный суффикс состоит из буквы "e" и следующего за ним порядка, выраженного в *ДЕСЯТИЧНОЙ* системе счисления. Число, определенное перед экспоненциальным суффиксом умножается на основание системы счисления (если оно явно не указано, то равно 10), возведенное в степень, определенную порядком.

Экспоненциальная форма	Десятичная
1.586e5	158600.0
1.586e-3	0.001586
8r3e2	192
2r11e6	192

Еще отметим, что система позволяет при выполнении вычислений смешивать числа, представленные в разных системах счисления, при этом результат, как правило, будет возвращаться в виде числа, представленного в десятичной системе счисления.

6.2 Классы `Float` и `Fraction`

Классы `Float` и `Fraction` позволяют представлять любые нецелые величины двумя разными способами. Класс `Float` представляет рациональные числа в 8-байтовом формате IEEE чисел с плавающей точкой в диапазоне от $\pm 4.19e - 307$ до $\pm 1.67e308$ и обеспечивают точность примерно в 18 значащих чисел. Некоторые примеры:

8.0 13.34 0.3 2.5e6 1.27e-30 -12.987654e12.

Экземпляры класса `Float` могут быть созданы или с помощью цифровой записи или получены как результат арифметических операций над экземплярами класса `Float`. Класс `Float` отвечает на сообщение `pi`, возвращая соответствующую константу.

Класс `Fraction` представляет рациональные числа, представляемые отношениями двух целых — числителя и знаменателя, знаменатель которого — неотрицательное целое число. Все арифметические операции с рациональными числами дают в результате приведенное рациональное число. Экземпляры класса `Fraction` получаются как результат арифметических операций, когда один из операндов — рациональная дробь, а второй — не является экземпляром класса `Float`. Рациональное число может быть получено и при выполнении операции деления над целыми числами, когда они не делятся нацело. Дополнительно к этому, протокол класса `Fraction` содержит метод класса с сообщением `numerator: numInteger denominator: denInteger`, который позволяет создавать рациональные числа с указанными числителем (`numInteger`) и знаменателем

(denInteger). Однако этого способа образования рациональной дроби лучше избегать, поскольку при таком образовании экземпляра класса `Fraction` не происходит приведения к дроби с взаимно простыми числителем и знаменателем. Об этом программист, создавая дробь таким образом, должен заботиться сам. Среди методов экземпляра существует метод `asFloat` для преобразования рациональной дроби в рациональное число с плавающей точкой.

Из всего сказанного следует, что класс `Fraction`, если не использовать при выполнении арифметических операций экземпляров класса `Float`, предоставляет достаточно редкую в языках программирования возможность вести выкладки без округлений и потому без потери точности. Чтобы стало ясно, как это достигается, приведем здесь с очевидными сокращениями почти полную реализацию класса `Fraction`.

Number subclass: #Fraction

instanceVariableNames: ' numerator denominator '

classVariableNames: ' '

poolDictionaries: ' '

Fraction class methods

numerator: n denominator: d

" Возвращает экземпляр класса Fraction с
числителем n и знаменателем d."

^self basicNew numerator: n denominator: d

Fraction methods

* aNumber

" Возвращает результат умножения приемника на число aNumber."

^(numerator * aNumber numerator) /
(denominator * aNumber denominator)

/ aNumber

" Возвращает результат деления приемника на число aNumber."

^(numerator * aNumber denominator) /
(denominator * aNumber numerator)

+ aNumber

"Возвращает результат сложения приемника и числа aNumber."

$$\frac{((\text{numerator} * \text{aNumber denominator}) + (\text{denominator} * \text{aNumber numerator}))}{(\text{denominator} * \text{aNumber denominator})}$$

... метод для сообщения "−" задается аналогично ...

// aNumber

"Возвращает целочисленный результат деления приемника на число aNumber с усечением к минус бесконечности."

$$\frac{(\text{numerator} * \text{aNumber denominator})}{(\text{denominator} * \text{aNumber numerator})}$$

\ \ aNumber

"Возвращает целочисленный остаток от деления приемника на число aNumber с усечением к минус бесконечности."

$$\frac{(\text{numerator} * \text{aNumber denominator}) \ \ }{(\text{denominator} * \text{aNumber numerator})}$$

= aNumber

$$\wedge (\text{aNumber isKindOf: Number}) \text{ and: } [\text{numerator} = \text{aNumber numerator} \text{ and: } [\text{denominator} = \text{aNumber denominator}]]$$

< aNumber

"Возвращает true, если приемник меньше числа aNumber, иначе возвращает false."

$$\wedge (\text{numerator} * \text{aNumber denominator}) < (\text{denominator} * \text{aNumber numerator})$$

... методы для сообщений <=, >, >= задаются аналогично ...

asFloat

"Возвращает приемник как число с плавающей точкой."

$$\wedge \text{numerator asFloat} / \text{denominator asFloat}$$

denominator

"Возвращает знаменатель приемника."

$$\wedge \text{denominator}$$

fraction

"Возвращает приемник за вычетом его целой части."

\wedge self — self truncated

negated

"Возвращает приемник с противоположным знаком."

\wedge Fraction

numerator: self numerator negated

denominator: self denominator

numerator

"Возвращает числитель приемника."

\wedge numerator

numerator: n denominator: d

"Частный. Возвращает приемник, определяя его числитель и знаменатель равными n и d, соответственно."

numerator := n.

denominator := d

printOn: aStream

"Добавляет ASCII-представление приемника в поток aStream."

numerator printOn: aStream.

aStream nextPut: \$.

denominator printOn: aStream

reciprocal

"Возвращает рациональную дробь, обратную к приемнику."

\wedge denominator / numerator

truncated

"Возвращает целую часть приемника."

\wedge numerator quo: denominator

6.3 Классы целых чисел

Класс **Integer** добавляет протокол для целых чисел. В нем три подкласса. Класс **SmallInteger** дает экономичное представление для

подмножества целых чисел, которые чаще всего используются в индексах и счетчиках. Диапазон представления для таких чисел от -16384 до 16383 . Большие целые, которые представляются экземплярами классов `LargePositiveInteger` и `LargeNegativeInteger` позволяют представлять целые числа длиной до $64K$ (в двоичном дополнительном коде). Заметим, что если результат арифметических операций над большими целыми представляется малым целым числом, то он в действительности будет малым целым.

В дополнение к сообщениям, наследуемым из класса `Number`, протокол класса `Integer` содержит методы-преобразования с сообщениями `asCharacter`, `asFloat`, метод для печати с сообщением `printOn: aStream base: anInteger`, и метод перечисления с сообщением `timesRepeat: aBlock`. Например, используя последнее сообщение, можно повторить вычисление блока `aBlock` целое число раз. Приведем такой пример, в котором блок не имеет аргументов:

```
a := 1.  
10 timesRepeat: [a := a + a].
```

Окончательное значение `a` — число 1024 , то есть 2 в степени 10 .

Класс `Integer` определяет три новых метода, которых нет его суперклассов.

Integer instance protocol

`factorial` — возвращает факториал приемника, который должен быть > 0 .

`gcd: anInteger` — возвращает наибольший общий делитель приемника и аргумента `anInteger`.

`lcm: anInteger` — возвращает наименьшее общее кратное приемника и аргумента `anInteger`.

Примеры применения этих сообщений

выражение	результат
-----------	-----------

3 factorial	6
55 gcd: 30	5
6 lcm: 10	30

В дополнение к операциям с целыми как с числами, некоторые алгоритмы используют тот факт, что можно интерпретировать целые как последовательность битов. Следующий протокол сообщений для манипуляции с битами определен в классе `Integer`.

`Integer` instance protocol

`bitAnd: anInteger` — возвращает экземпляр класса `Integer`, все биты которого получены как результат выполнения операции логического И над соответствующими битами приемника сообщения и аргумента `anInteger`.

`bitOr: anInteger` — возвращает экземпляр класса `Integer`, все биты которого получены как результат выполнения операции логического ИЛИ над соответствующими битами приемника сообщения и аргумента `anInteger`.

`bitXor: anInteger` — возвращает экземпляр класса `Integer`, все биты которого получены как результат выполнения операции логического ИСКЛЮЧАЮЩЕГО ИЛИ над соответствующими битами приемника сообщения и аргумента `anInteger`.

`bitAt: index` — возвращает бит приемника в позиции `index`.

`bitInvert` — возвращает экземпляр класса `Integer`, все биты которого дополнены соответствующим битам приемника.

`bitShift: anInteger` — возвращает экземпляр класса `Integer`, чье значение (в бинарном представлении) получается сдвигом приемника сообщения (в бинарном представлении) на число битов, указанных аргументом `anInteger`. Если `anInteger < 0`, сдвиг производится вправо, а если `anInteger > 0` — влево.

Вот несколько примеров. Заметим, что по умолчанию основание системы счисления при выводе чисел, принимается равным 10.

выражение	результат
2r111000111000111	29127
2r101010101010101	21845
2r101000101000101	20805
2r000111000111000	3640
29127 bitAnd: 3640	0
29127 bitOr: 3640	32767
32767 radix:2	2r111111111111111
29127 bitOr: 21845	30167
3640 bitShift: 1	7280

7 Класс Random

Класс **Random** не входит в иерархию класса **Magnitude**, но часто его экземпляры необходимы во многих приложениях, требующих выбора случайного числа. Случайные числа нужны, например, в статистических приложениях и в алгоритмах шифровки данных. Класс **Random** описывает генератор случайных чисел и включается в иерархию класса **Stream**, которую мы подробно рассмотрим в одной из наших следующих методических разработок. Здесь же только кратко остановимся на классе **Random**, поскольку он тесно связан с числами и обеспечивает простой способ получения последовательности случайных чисел, равномерно распределенных на интервале (0,1).

Экземпляр класса **Random** — источник, порождающий следующее случайное число. Этот источник создается с помощью выражения `rand := Random new`. Когда потребуется следующая случайная величина, экземпляру класса **Random** надо послать сообщение `next`, то есть вычислить выражение `rand next`, которое вернет вещественное число между 0.0 и 1.0. Реализация метода, соответствующего сообщению `next`, опирается на линейный конгруэнтный метод Д.Х. Лемера в той форме, как он изложен в книге *Д. Кнута, Искусство программирования для ЭВМ, т.2, М., "Мир", 1977, стр. 22.*

```

next
| temp |
"Линейный конгруэнтный метод Д.Х.Лемера."
[seed <- 13849 + (27181 * seed) bitAnd: 8r177777.
temp <- seed / 65536.0.
temp = 0] whileTrue: [ ].
^temp

```

Можно послать экземпляру класса `Random` и сообщение `next`: `anInteger`, чтобы получить упорядоченный набор из `anInteger` случайных чисел, и сообщение `nextMatchFor: aNumber`, чтобы определить равно ли следующее случайное число числу `aNumber`. Как и почему все так происходит, можно узнать познакомившись подробнее с классом `Stream` и его подклассами.

А пока предположим, что нам необходимо случайным образом получить одно из 10 первых натуральных чисел, то есть получить число из набора 1,2,...,10. Этого можно добиться, вычисляя выражение

$$(\text{rand next} * 10) \text{ truncated} + 1.$$

Теперь рассмотрим процесс вычисления этого выражения по шагам

выражение	результат
<code>rand next</code>	случайное число между 0.0 и 1.0
<code>rand next * 10</code>	случайное число между 0.0 и 10.0
<code>(rand next * 10)truncated</code>	целое число из набора 0,1,2,...,9
<code>(rand next * 10) truncated + 1</code>	целое число из набора 1,2,...,10

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

Содержание

1	Класс Magnitude	3
2	Ассоциативные пары	4
3	Класс Character	6
4	Класс Date	10
5	Класс Time	13
6	Протокол числовых классов	15
6.1	Протокол класса Number	16
6.2	Классы Float и Fraction	24
6.3	Классы целых чисел	27
7	Класс Random	30
	Литература	32

Ю.А. Кирютенко <jakir@mmf.unird.ac.ru>
В.А. Савельев <vasav@mmf.unird.ac.ru>