

Ю.А. Кирютенко, В.А. Савельев

Объектно-ориентированное программирование  
и язык Smalltalk

Общие концепции и синтаксис

Ростов-на-Дону

1997

# Объектно-ориентированное программирование и язык Smalltalk

## Общие концепции и синтаксис

### Аннотация

Методическая разработка посвящена современному направлению в программировании — объектно-ориентированной методологии программирования и языку Smalltalk и предназначена для студентов 3–5 курсов механико-математического факультета и слушателей ФПК.

Печатается в соответствии с решением кафедры математического анализа Ростовского государственного университета, протокол № 4 от 5 декабря 1995 года.

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

© 1995, 1997, Ю.А. Кирютенко, В.А. Савельев

# 1 Объектно-ориентированный подход в программировании

В последнее время развитие аппаратных средств существенно опережает развитие систем и средств программирования. Чтобы выправить положение, в 70-80-х годах были предложены различные подходы к увеличению производительности труда программиста. Среди этих попыток выделяется такое популярное направление, как объектно-ориентированный подход к конструированию и кодированию программ. Особую роль в популярности этого подхода сыграло как его тесная связь с интерфейсами пользователя (особенно графическими), так и включение элементов этого подхода в популярные (на персональных компьютерах фирмы IBM) реализации гибридных языков программирования C++ и Pascal with Objects фирмы Borland.

До сих пор большинство используемых программных систем построены на принципах структурного подхода, суть которого состоит в декомпозиции системы на ряд модулей, процедур, функций и структур данных, связанных общим алгоритмом функционирования. Но распространение мощных персональных компьютеров (сравнимых с рабочими станциями 70–80-х годов) создало в 90-х годах основу для широкого применения объектно-ориентированного подхода на практике. В последнее время более широко начинают использоваться языки программирования, созданные в рамках объектно-ориентированной методологии, такие как Smalltalk и Java.

Новая методология ориентирована, прежде всего, на создание больших систем, коллективную их разработку, последующее их активное сопровождение в эксплуатации и регулярные модификации. Среди типовых задач, для которых ООМ является перспективной, можно выделить такие:

- автоматизация эксперимента, робототехника;
- диспетчеризация, планирование;
- интерфейс пользователя, анимация;
- коммуникации, связь;
- медицина, экспертные системы;
- обработка коммерческой информации;

- операционные системы;
- системы управления;
- тренажеры, моделирование.

В основе объектно-ориентированной методологии программирования (ООМ) лежит объектный подход, когда прикладная предметная область представляется в виде совокупности объектов, которые взаимодействуют между собой посредством передачи сообщений. Под объектом понимается некоторая сущность (реальная или абстрактная) конкретной предметной области, обладающая состоянием, поведением и индивидуальностью. Состояние объекта характеризуется перечнем всех его возможных (обычно статических) свойств и значениями каждого из этих свойств (обычно динамических). Состояние объекта описывается его переменными. Поведение объекта (или его функциональность) характеризует то, как объект взаимодействует с другими объектами или подвергается взаимодействию других объектов, проявляя свою индивидуальность. Индивидуальность — это такие свойства объекта, которые отличают его ото всех других объектов. Поведение объекта реализуется в виде функций, которые называют методами. При этом структура объекта доступна только через его методы, которые в совокупности формируют интерфейс объекта.

Такой подход позволяет локализовать принимаемые решения рамками объекта, объединяя в нем и структуру и поведение, а следовательно, снижая сложность отдельной программы (реализующей объект). Эта идея объединения структуры и поведения в одном месте и сокрытия всех данных внутри объекта, что делает их невидимыми для всех, за исключением методов самого объекта называется инкапсуляцией. Это позволяет объектам функционировать совершенно независимо друг от друга, скрывая за интерфейсом детали реализации. Инкапсуляция позволяет рассматривать объекты, как изолированные "черные ящики", которые знают и умеют выполнять определенные действия. С этой точки зрения, внутреннее устройство "черных ящиков" для нас значения не имеет, нам все равно, что происходит внутри. Важно только знать, что надо положить в ящик при обращении к нему и что мы при этом из него получим. Таким образом, объекты объектно-ориентированных систем — это минимальные единицы инкапсуляции.

Но к объекту может обращаться не только программист, скрывающийся под местоимением **МЫ**, но и любой объект, функционирующий в системе. Для этого нужно только послать интересующему объекту сообщение, которое представляет просьбу (приказ, требование) выполнить некоторые действия. И если такая просьба может быть выполнена принявшим сообщение объектом, то она выполняется, а если не может быть выполнена, то не выполняется, и в какой-то форме пославший сообщение объект информируется о реакции на полученное сообщение.

Но как управлять таким миром объектов, когда их становится достаточно много? Ведь многие из них будут очень сильно отличаться друг от друга, как например, объекты, описывающие принтер и черепаху Тортиллу, а другие объекты будут очень похожи друг на друга, как например, объекты, описывающие автомобиль марки "Форд" и автомобиль марки "Оппель". Здесь на сцену выходит одна из ключевых концепций объектно-ориентированного программирования — идея группировки объектов в классы, в соответствии с тем как они устроены и действуют. Такая идея впервые была реализована еще в 60-ые годы в языке Simula. Под классом понимается множество объектов, связанных общностью структуры и поведения. Таким образом, класс можно сравнить с шаблоном, по которому создаются объекты. Именно класс вначале описывает переменные и методы объекта, то есть структуру и поведение объекта, и определяет механизмы создания реально существующего в системе объекта, который, когда создается, представляет собой экземпляр класса.

Наведение с помощью классов порядка в мире объектов — большое достижение, но можно пойти дальше, определяя некоторый порядок и среди классов. Достигается это с помощью введения механизма наследования — пожалуй, самого мощного средства в любой объектно-ориентированной системе, поскольку оно позволяет многократно использовать однажды созданный код. Механизм наследования очень прост: один класс, называемый в рамках этих отношений суперклассом, полностью передает другому классу, который называется подклассом, свою структуру и поведение, то есть все свои переменные и все методы. Что далее делать с этим богатством определяет только подкласс: он может добавить в структуру что-то свое, что-то из наследуемого интерфейса он может использовать без изменений, что-то изменить, и, разумеется, может добавить свои собственные методы. То есть класс с помощью подклассов расширя-

ется, и как результат, создаваемые объекты становятся все более и более специализированными. Классы, расположенные по принципу наследования, начиная с самого общего, базового класса, образуют иерархию классов.

Разумеется система, реализующая такие принципы построения, предъявляет более жесткие, чем при структурном подходе, требования к производительности вычислительной системы.

Прежде, чем двигаться дальше, стоит отметить, что ООМ не является отрицанием или противопоставлением структурному подходу, напротив, методология структурного программирования входит составной частью в ООМ как средство программирования структуры и поведения самих объектов. Эту методологию правильнее представлять как инструмент, позволяющий снизить сложность задачи и подойти к созданию таких систем, поведение которых невозможно представить в виде исчерпывающего набора всех возможных ситуаций и разветвлений алгоритма. Сегодня теоретически обоснована и практически доказана возможность создания на основе ООМ проектов высокой степени сложности, включающих миллионы строк кода. ООМ прекрасно реализуется на обычных персональных компьютерах, предоставляя при этом целый ряд новых преимуществ по сравнению с процедурным подходом. Но вот на характер мышления программиста и дисциплину проектирования программного продукта новая методология, безусловно накладывает свой отпечаток, особенно на первых этапах использования ООМ.

После краткого описания некоторых ключевых понятий объектно-ориентированной методологии, можно сделать вывод, что концептуально объектно-ориентированная методология программирования опирается на объектный подход, который включает в себя четыре основных принципа:

**Абстрагирование.** Выделение таких существенных характеристик объектов, которые отличают его ото всех других объектов и которые четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. Только существенное для данной задачи и ничего более. Минимальной единицей абстракции в ООМ является класс.

**Ограничение доступа.** Процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта, как целого.

**Модульность.** Свойство системы, связанное с возможностью декомпозиции на ряд тесно связанных частей (модулей). Модульность опирается на дискретное программирование объектов, которые можно модернизировать или заменять, не воздействуя на другие объекты и систему в целом.

**Существование иерархий.** Ранжирование, упорядочивание по некоторым правилам объектов системы.

Объектно-ориентированная методология (ООМ) включает в себя такие компоненты:

- объектно-ориентированный анализ (ООА),
- объектно-ориентированное проектирование (OOD),
- объектно-ориентированное программирование (ООР).

ООА — методология анализа сущностей реального мира на основе понятий класса и объекта, составляющих словарь предметной области, для понимания и объяснения того, как они (сущности) взаимодействуют между собой.

Рассматривая реальную задачу, аналитик разбивает ее на некоторое число предметных областей. Каждая предметная область — мир, наделенный объектами. В предметной области выделяются классы объектов, которые, если это необходимо, разбиваются на подклассы. Каждый класс и его подкласс анализируются в три этапа: информационное моделирование, моделирование состояний, моделирование процессов.

Модели ООА в дальнейшем преобразуются в объектно-ориентированный проект. OOD — методология проектирования программного продукта, соединяющая в себе процесс объектной декомпозиции, опирающийся на выделение классов и объектов, и приемы представления моделей, отражающих логическую (структура классов и объектов) и физическую (архитектура моделей и процессов) структуру системы. Фундаментальные понятия OOD: инкапсуляция, наследование, полиморфизм.

**Инкапсуляция.** Концепция сокрытия в как бы ”капсуле” всей информации об объекте, то есть объединение в некое целое данных и процедур (методов) их обработки. Единицей инкапсуляции в OOD является объект, в котором содержатся и данные

состояния объекта и сообщения, которые объект может обрабатывать.

**Наследование.** Получение от предшественника — такое соотношение между классами, находящимися в некоторой определенной иерархии, при которой один класс моделирует поведение и свойства другого класса, добавляя свою специфику. Класс поведение которого наследуется называется суперклассом, а класс, который наследует поведение, называется подклассом.

**Полиморфизм.** Возможность единообразного обращения (посылки объектам одноименных сообщений) при сохранении уникального поведения объектов. Другими словами, поскольку поведение объектов определяется методами, метод, ассоциированный с одним и тем же именем сообщения, допускает различные реализации для разных классов.

Созданный проект превращается в программный продукт в процессе объектно-ориентированного программирования — такой методологии программирования, которая основана на представлении программного продукта в виде совокупности объектов, каждый из которых является слепком (экземпляром) определенного класса, а классы образуют иерархию на принципах наследования. Таким образом, при объектно-ориентированном подходе исчезает понятие исполняемой программы. Решение поставленной задачи сводится к построению необходимых классов, и управлению создаваемыми ими объектами-экземплярами.

Фундаментальная концепция ООР состоит в том, что объекты и классы взаимодействуют друг с другом путем передачи сообщений. Для этого необходимо, чтобы объекты определялись вместе с сообщениями, на которые они реагируют, в отличие от процедурного стиля программирования, когда сначала определяются данные, которые затем передаются в процедуры (функции) как параметры. При этом средством программирования выступает один из объектно-ориентированных языков программирования.

Язык программирования называется *объектно-ориентированным*, если

- есть поддержка объектов как абстракций данных, имеющих интерфейсную часть в виде поименованных операций, и защищенную область локальных данных;



- все объекты относятся к соответствующим типам (классам);
- классы могут наследовать от суперклассов.
- любые данные хранятся как объекты, размещаемые с автоматическим выделением и освобождением памяти. Объект существует в системе до тех пор, пока его можно именовать.

Последний принцип отличает чистые объектно-ориентированные языки такие как **Smalltalk**, **CLOS** (**Common Lisp Object System**), а в среде **Windows** — язык **Actor**, от гибридных языков программирования, выросших из ранее существовавших процедурных языков (**Object Pascal**, **C++**). Эти подходы — как бы крайности в семействе объектно-ориентированных языков. Ближе к середине лежит совершенно новый, полностью построенный на принципах объектно-ориентированной идеологии, но все же позволяющий нарушать последний принцип, язык **Java**.

## 2 Язык программирования Smalltalk

### 2.1 Краткая история языка

С объектно-ориентированной методологией мы познакомимся, изучая язык программирования **Smalltalk**, при создании которого, как программного продукта, ООМ была полностью реализована. Разработка языка **Smalltalk** началась в 1970 году в исследовательском центре фирмы **Xerox**. Идея создания однородной объектно-ориентированной системы типа **Smalltalk** принадлежит Алену Кею (**Alan Kay**). Работа над языком продолжалась почти 10 лет. Главным архитектором проекта все эти годы был Дэн Ингалс (**Dan Ingalls**). Значительный вклад в создание языка внесли также Питер Дейтч (**Peter Deutsch**), Глен Краснер (**Glenn Krasner**), Ким МакКолл (**Kim McCall**). Прекрасное изложение основ языка и ООМ, отражающее эту работу, содержится в книге [1], написанной членами группы разработчиков языка **Smalltalk**. Эта книга в компьютерном мире известна как Голубая Книга.

Существовали следующие реализации языка **Smalltalk**, созданные фирмой **Xerox Park**: **Smalltalk-72**, **-74**, **-76**, **-78**, **-80** (цифры означают год создания). Последняя версия, как коммерческий продукт, появилась на рынке программного обеспечения в 1983 году и приобрела достаточно широкую известность. Реализации **Smalltalk-80**

сегодня существуют для многих классов ЭВМ и в первую очередь для персональных компьютеров и рабочих станций. В конце 80-х появилась версия **Smalltalk-80** в среде Windows под названием **Object Works**.

Существует коммерческая версия языка для IBM AT под DOS, созданная в конце 80-х годов фирмой **Digitalk, Inc.** Это **Smalltalk/V**. В связи с ограниченностью ресурсов на ранних IBM-компьютерах и их клонах, эта система содержит не все классы, поставляемые со **Smalltalk-80**. Однако основные системные классы и большинство примитивов в эту систему включены. Кроме того сделаны расширения графического ядра, которые позволяют работать с цветными графическими адаптерами, доступными на IBM-совместимых компьютерах. В начале 90-х той же фирмой создана версия **Smalltalk/V for Windows**.

В начале 90-х годов в Институте проблем информатики Российской Академии Наук была произведена адаптация языка **Smalltalk/V** для русскоязычного пользователя, после которой в созданной среде программирования стало возможным писать программы на "русском языке". Поскольку адаптация была ориентирована на совсем уж маломощные ЭВМ советского производства, **Smalltalk/V** был несколько упрощен (например, исчезла многозадачность), была изменена структура некоторых классов и их иерархия, но зато система вполне работоспособна на IBM-совместимых компьютерах даже при наличии 512Mb оперативной памяти. И хотя время таких машин прошло, их еще достаточно много в средних и даже высших учебных заведениях. Очень привлекательной является предоставляемая этой адаптацией возможность познакомиться с основными чертами новой идеологии, даже тем, кто плохо знает (или совсем не знает) английский язык.

Тогда же, в начала 90-х годов, на базе языка **Smalltalk-80** возникли объектно-ориентированные системы, содержащие инструменты для визуального программирования, такие как **Enfin** и **VisualAge for Smalltalk**, которые настраиваются под несколько операционных систем. Версия 3.0 системы **VisualAge for Smalltalk** фирмы IBM требует, например, компьютера по крайней мере с 24 Mb оперативной памяти и 300 Mb на винчестере, но предоставляет в распоряжение программистов мощные и разнообразные инструменты как индивидуальной, так и коллективной разработки программного продукта.

Основы методологии и языка мы будем рассматривать опираясь на систему **Smalltalk/V**. Она работает на достаточно распространен-

ных IBM-совместимых ЭВМ мощности не ниже PC/AT286+ и содержит почти все, что необходимо для первоначального знакомства с новым направлением.

## 2.2 Иерархия классов Smalltalk и ее особенности

Object

Behavior

Class

Metaclass

BitBlt

CharacterScanner

Pen

Animation

Commander

BitEditor

Boolean

False

True

ClassBrowser

ClassHierarchyBrowser

ClassReader

Collection

Bag

IndexedCollection

FixedSizeCollection

Array

CompiledMethod

BitMap

BitArray

FileHandle

Interval

String

Symbol

OrderedCollection

Process

SortedCollection

Set

Dictionary

IdentityDictionary

MethodDictionary  
SystemDictionary  
SymbolSet  
Compiler  
    LCompiler  
Context  
    HomeContext  
CursorManager  
    NoCursorManager  
DemoClass  
DeleteClass  
Directory  
DiskBrowser  
Dispetcher  
    PointDispetcher  
    ScreenDispetcher  
    ScroolDispetcher  
        GraphDispetcher  
            FormEditor  
            ListSelector  
            TextEditor  
                PromptEditor  
    TopDispetcher  
DispatchManager  
DisplayObject  
    DisplayMedium  
        Form  
            BiColorForm  
            ColorForm  
            DisplayScreen  
                ColorScreen  
Dos  
EmptySlot  
File  
Font  
InputEvent  
Inspector  
    Debugger  
    DictionaryInspector  
Icon

- Magnitude
  - Association
  - Character
  - Date
  - Number
    - Integer
      - LargeNegativeInteger
      - LargePositiveInteger
      - SmallInteger
    - Float
    - Fraction
  - Time
- Menu
- MethodBrowser
- Message
- Pane
  - SubPane
    - GraphPane
    - IconPane
    - ListPane
    - TextPane
  - TopPane
- Pattern
  - WildPattern
- Point
- ProcessScheduler
- Prompter
- Rectangle
- Semaphore
- Stream
  - ReadStream
    - Random
  - WriteStream
    - ReadWriteStream
    - FileStream
    - TerminalStream
- StringModel
- Text
- TextSelection
- UndefinedObject

Как видно из представления, все классы языка являются под-классами класса **Object**, который называется базовым для иерархии классов и в котором определены общая для всех объектов структура и поведение. Класс **Object** не имеет суперкласса, но чтобы сохранить единство построения системы, принимается, что его суперклассом является специальный неопределенный объект `'nil'`.

Как уже отмечалось — **Smalltalk** однородный объектно-ориентированный язык, в котором все объекты являются экземплярами классов, а единственным способом управления объектами является посылка им сообщений. Например, чтобы определить, экземпляром какого класса является некоторый объект, надо послать этому объекту сообщение `class`, которое требует от объекта вернуть имя класса, по шаблону которого он создан. Чтобы сделать это, необходимо, согласно синтаксиса языка, сначала указать объект, которому посылается сообщение, а затем записать само сообщение. Например, мы желаем узнать имя класса для объекта 3.2. В этом случае получится выражение

### 3.2 class

В ответ на принятое сообщение объект 3.2 вернет имя класса **Float**.

Но как в систему попадают экземпляры классов? Их надо создавать согласно шаблонов, определяемых классами! Следовательно, чтобы создать экземпляр класса надо обратиться к классу с сообщением, требующим создания экземпляра. Значит сами классы уже выступают как объекты системы. И, подобно тому как экземпляры создаются из классов, служащих для них моделью или образцом, так и сами классы должны рассматриваться в языке как объекты, создаваемые в соответствии с шаблоном, заключенном в определенном классе. Такой класс называется метаклассом данного класса. Он создается автоматически при создании класса. При этом класс — единственный экземпляр своего метакласса. Такой метакласс не имеет имени и доступ к нему осуществляется посылкой уже известного нам сообщения `class` к классу. Например, пошлем сообщение `class` классу **Float**, то есть запишем выражение вида `Float class`. В ответ на принятое сообщение класс **Float** должен вернуть имя класса, экземпляром которого он является, то есть он должен вернуть имя

своего метакласса, но поскольку у метаклассов нет имен, будет возвращено `Float class`.

Но сами метаклассы — это классы системы. Какое они в ней занимают место? Первое и совершенно очевидное правило состоит в том, что приведенная выше иерархия классов сохраняется и для их метаклассов в том смысле, что если `класс_1` есть подкласс `класса_2`, то и метакласс `класса_1` есть подкласс метакласса `класса_2`. Второе правило вытекает из общего принципа построения системы, согласно которому любой метакласс в системе, являясь ее классом, должен быть экземпляром некоторого класса. Оно состоит в том, что любой метакласс в системе есть экземпляр класса `Metaclass`, который для них, являясь шаблоном, описывает их наиболее общие свойства, связанные с поведением в системе тех объектов, которые умеют создавать по одному экземпляру, представляющему класс.

Для завершения построения системы, остается только связать между собой иерархию классов и метаклассов. Чтобы это сделать, вспомним о том, что в иерархии классов базовым классом является класс `Object`, который сам не имеет суперкласса. Но класс `Object`, согласно предыдущего, имеет свой метакласс `Object class`, который, с одной стороны есть экземпляр класса `Metaclass`, как и все другие метаклассы системы, а с другой стороны, этот класс является "базовым" классом иерархии метаклассов, и согласно принципа наследования при построении иерархии, определяет общую структуру и поведение всех экземпляров метаклассов системы, то есть классов. Отражая эту роль метакласса `Object class`, в системе `Smalltalk` создан класс с именем `Class`, который является суперклассом для `Object class`. Таким образом, все метаклассы, как объекты системы, являются подклассами класса с именем `Class`, одновременно являясь экземплярами класса `Metaclass`. Таким образом, два класса системы, а именно классы `Class` и `Метакласс` описывают структуру и поведение тех объектов системы, которые могут создавать экземпляры. Все то общее, что присуще классам и метаклассам системы, описывается в классе `Behavior`, для которого классы `Class` и `Метакласс` являются подклассами. Кроме того, класс `Behavior` определяется всю информацию, необходимую интерпретатору `Smalltalk`'а, которая включает в себя полное описание существующей в системе иерархии классов и общие методы, которые позволяют:

- создавать словари методов, экземпляры классов, иерархию классов;





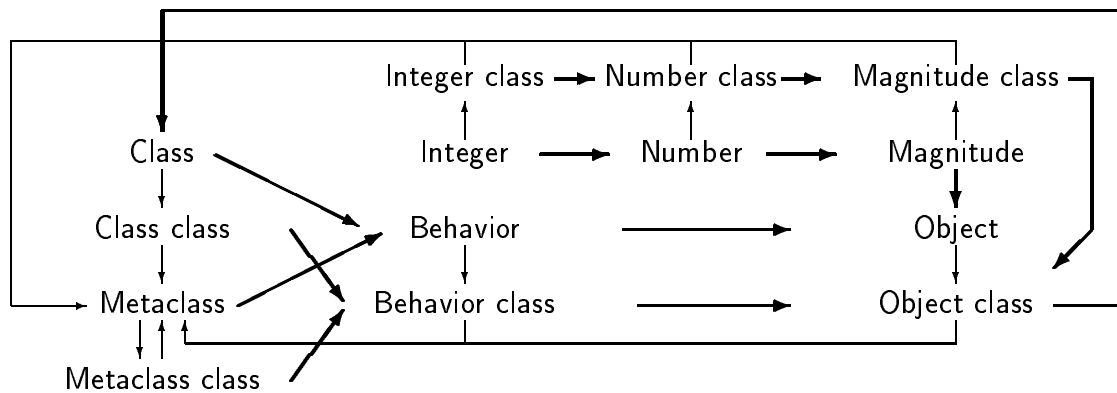


Рис. 2. Взаимосвязь иерархий классов и метаклассов  
 $A \rightarrow B$  —  $A$  есть подкласс  $B$ ;  $C \rightarrow D$  —  $C$  есть экземпляр  $D$

- осуществлять доступ к содержимому словарей методов, к экземплярам и переменным (экземпляра, класса, пула), к иерархии классов;
- проверять содержимое словарей методов, форму экземпляров;
- перечислять существующие подклассы и экземпляры.

Таким образом, в системе **Smalltalk** имеется не одна, а две иерархии — одна для классов, а вторая для метаклассов этих классов. Причем вторая иерархия полностью соответствует первой и обе иерархии между собой тесно связаны через класс системы **Smalltalk**, который называется **Class**. Особое место занимает в системе класс **Metaclass**, который, в силу своего определения в системе, порождает метакласс **Metaclass class** как свой экземпляр, а с другой стороны, в силу определения метакласса, сам является экземпляром собственного метакласса **Metaclass class** (см. Рис. 2).

Все это необходимо знать для правильного понимания функционирования системы.

С самого начала имена классов системы не случайно писались с прописной буквы. Все они являются именами глобальных переменных (то есть именами переменных, доступных всем объектам системы), содержатся в системном словаре с именем **Smalltalk** и могут вызываться по имени в любое время и в любом месте. Более подробно все это будет рассмотрено дальше в разделе 3.3 “Переменные в языке **Smalltalk**”.

## 2.3 Определение класса и пример класса

Продолжая изучение классов и их иерархии, рассмотрим протокол класса. Этот протокол необходим, например, при создании нового класса. Прежде всего, в протоколе нужно точно указать место создаваемого класса в иерархии ‘класс-суперкласс’. Затем, необходимо описать как структуру самого класса, как компонента системы, так и структуру его экземпляра. После этого надо определить интерфейс самого класса (то есть записать те методы, которые должен уметь выполнять сам класс как компонент системы) и интерфейс его экземпляра (то есть записать те методы, которые должен уметь выполнять экземпляр данного класса). Но класс является экземпляром своего метакласса и по нашим правилам именно метакласс ответственен за то, как ведет себя его экземпляр. Следовательно, определяя интерфейс самого класса, мы описываем и метакласс данного класса, который будет создан автоматически по протоколу создаваемого класса, и автоматически займет подходящее место в иерархии метаклассов.

В переводе на русский язык, протокол класса в Smalltalk’е выглядит следующим образом:

```
<Имя_суперкласса> <тип_подкласса>: <Имя_класса>  
  именаПеременныхЭкземпляра: 'имя1 имя2 ... '  
  именаПеременныхКласса: 'Имя1 Имя2 ... '  
  пулы: 'Имя1 Имя2 ... '
```

```
!<имя_класса> методы класса!  
  метод_1 !  
  ...  
  ...  
  ...  
  метод_N !!
```

```
!<имя_класса> методы экземпляра!  
  метод_1 !  
  ...  
  ...  
  ...  
  метод_S !!
```

Напомним, что, как и во многих других языках программирования, под именем переменной всегда понимается последовательность букв и цифр, начинающаяся с буквы. Прежде чем подробно рассмотреть протокол класса и все в нем объяснить, приведем большую часть описания системного класса `Association`, который и послужит нам далее в качестве иллюстрации. Комментарии к методам, которые предназначены для программиста и представляют краткую характеристику метода, но никак не влияют на исполнение метода в системе, переведены на русский язык. Весь остальной текст — неперебиваемые, синтаксически правильные выражения языка, за исключением знаков '!', которые используются как разделители при выводе протокола в текстовой форме.

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

!Association class methods!

```
key: aKey
  "Создает и возвращает экземпляр класса Association,
   с ключом равным заданному объекту aKey."
  ^self new key: aKey!
```

```
key: aKey value: anObject
  "Создает и возвращает экземпляр класса Association
   с заданными ключом aKey и значением anObject."
  ^(self key: aKey) value: anObject! !
```

!Association methods !

```
= anAssociation
  "Возвращает значение true (истина), если ключ приемника
   равен ключу заданной пары anAssociation, в противном
   случае возвращает значение false (ложь)."
  ^(anAssociation class == Association)
   and: [key = anAssociation key]!
```

hash

"Возвращает целое число, представляющее собой хэш—значение ключа приемника."

^ key hash!

key

"Возвращает ключ (первый аргумент) приемника"

^ key!

key: anObject

"Задаёт ключ (первый аргумент) приемника.

Возвращает приемник в качестве результата."

key := anObject!

printOn: aStream

"Записывает символьное представление приемника в заданный поток aStream."

key printOn: aStream.

aStream nextPutAll: ' ==> '.

value printOn: aStream!

storeOn: aStream

"Записывает символьное представление приемника в заданный поток aStream, из которого затем возможно однозначно восстановить приемник."

aStream nextPutAll: 'Association key: ('.

key storeOn: aStream.

aStream nextPutAll: ') value: ('.

value storeOn: aStream.

aStream nextPut: \$)!

value

"Возвращает значение (второй аргумент) приемника"

^ value!

value: anObject

"Задаёт значение (второй аргумент) приемника.

Возвращает приемник в качестве результата."

value := anObject! !

Итак, при определении класса `Association` первая строка имеет вид:

`Magnitude subclass: #Association`

Это значит, что класс `Association` есть подкласс класса `Magnitude`.

Вообще в `SmalltalkV` тип подкласса может вводиться одним из следующих трех способов:

- 1) `subclass:`
- 2) `variableSubclass:`
- 3) `variableByteSubclass:`

Эти ключевые слова определяют подклассы, чьи экземпляры содержат, соответственно:

- 1) именованные переменные экземпляра;
- 2) именованные и индексированные переменные экземпляра;
- 3) индексированные переменные экземпляра, представляющие массив байтов.

Третье сообщение определяет подкласс, чьи экземпляры содержат области памяти, организованные как массивы байтов, что позволяет эффективно хранить данные. Такие объекты определяют элементарные значения данных, и таковыми, например, являются экземпляры классов `String` и `Symbol`. Доступ к байтам этих классов осуществляют те же примитивные сообщения, что для объектов с индексированными переменными, но они всегда возвращают или присваивают целые значения, способные разместиться в байте.

Переменные экземпляра определены внутри каждого экземпляра класса, уникальны и прямые ссылки на них из других экземпляров этого класса, из других классов или экземпляров других классов невозможны. Эти переменные доступны только тому экземпляру, в котором они определены. Имена этих переменных пишутся со строчной буквы. В экземпляре класса `Association` две переменные: `key`, `value`.

Переменные класса являются глобальными переменными, их имена пишутся с прописной буквы. Эти переменные доступны всем экземплярам данного класса и всем экземплярам подклассов данного класса, а также самому классу и всем его подклассам как объектам. В классе `Association` переменных класса нет.

Пулы — это словари, состоящие из переменных. Этими переменными могут пользоваться те классы и экземпляры тех классов, в определении которых эти пулы указаны. Например, пул **CharacterConstants** или пул **FunctionKeys**. Как видим из определения, в классе **Association** пулов нет. Примером пула является словарь системы (экземпляр класса **SystemDictionary**) с именем **Smalltalk**, уже упоминавшийся выше и содержащий все глобальные переменные системы.

Подкласс наследует от суперкласса все его переменные экземпляра, класса и все его пулы, то есть экземпляр подкласса имеет все переменные экземпляра, определенные в суперклассе, и все переменные экземпляра, определенные в подклассе, а также имеет доступ ко всем переменным класса и пулам, определенным как в суперклассе, так и в подклассе. Поскольку при определении класса **Magnitude** списки переменных класса и экземпляра пусты и нет пулов, то этот суперкласс не добавляет переменных в класс **Association** и в его экземпляры. Но класс **Magnitude** — подкласс базового класса **Object**, в котором нет пулов и переменных экземпляра, но есть три переменных класса: **RecursionInError**, **Dependents**, **RecursiveSet**, следовательно, сам класс **Association** как объект системы и все его экземпляры могут обращаться к этим переменным.

Все, о чем мы так долго говорили, содержится в выражении

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

которое представляет собой посылку сообщения с селектором

```
subclass:instanceVariableNames:classVariableNames:poolDictionaries:
```

объекту системы, в данном случае классу **Object**, и как реакция на это сообщение происходит определение в системе нового класса с именем **Association**, с подходящей сообщению структурой как самого класса, так и его будущих экземпляров.

Наверное в последнем предложении не все понятно, но ведь мы не все еще и разобрали. В конце концов все прояснится. Двигаемся дальше!

## 2.4 Сообщения и методы в языке Smalltalk

Метод класса или метод экземпляра состоит из шаблона сообщения и тела метода. Шаблон сообщения состоит из имени сообщения (или, как часто говорят, селектора сообщения) и формальных объектов-аргументов, если они есть. Так шаблон сообщения `key: aKey value: anObject` состоит из имени сообщения `key:value:` и двух формальных объектов-аргументов `aKey` и `anObject`. Тело метода содержит выражения языка, которые составляют программу, выполняемую объектом-приемником по сообщению. Каждый класс имеет набор методов, определяющих сообщения двух типов: методы класса, понятные классу, как объекту системы, и методы экземпляра, понятные экземпляру этого класса. Методы класса выполняют сообщения, посланные классу, который является экземпляром своего метакласса, поэтому такие сообщения выполняются этим метаклассом. Методы класса обычно включают сообщения для создания экземпляров класса (например, `new:`), инициализации переменных класса. Приемником таких сообщений может быть только класс, а не экземпляр класса. Так, например, классу `Association` можно послать сообщение вида (то есть сообщение с шаблоном)

`key: aKey value: anObject`

по которому класс `Association` создаст экземпляр класса с заданными в сообщении ключом и значением. Например,

`myIndex := Association key: 'Index' value: 344017`

В результате, как реакция класса на это сообщение, будет создан экземпляр данного класса с именем `myIndex` с заданными значениями своих переменных. `:=` — это символ операции присвоения, после ее выполнения переменная с именем `myIndex` будет указывать на созданный экземпляр класса `Association`.

Методы экземпляра выполняют сообщения, посылаемые экземплярам данного класса, например

`myIndex value.`

Здесь сообщение с селектором `value` посылается экземпляру клас-

са `Association` с именем `myIndex`, при этом будет возвращено число 344017. Но почему так получилось? С последним сообщением почти все ясно. Чтобы понять, что происходит, достаточно посмотреть на определение класса `Association` и поискать там метод с шаблоном сообщения `value`. Там такой метод есть и состоит он всего из одного выражения, `^value`, которое и выполняется. Символ `^` — символ возврата значения, и когда он встречается в теле метода, результат вычисления стоящего после него выражения возвращается как результат выполнения метода. Но взглянув на методы в классе `Association` мы обнаруживаем, что символ возврата значения присутствует в теле не каждого метода. Что же будет возвращаться как результат выполнения метода в этом случае? Общее правило здесь таково: если нет указаний, что возвращать, по умолчанию возвращается приемник сообщения. Запомните это!

Вернемся к первому сообщению, с помощью которого создавали экземпляр класса. Чтобы проследить за процессом выполнения метода, соответствующего имени сообщения `key:value:`, остановимся на более подробно на механизмах поиска по сообщению необходимого метода и механизмах выполнения найденного метода. Итак, как уже отмечалось, выполнение любого действия в `Smalltalk`'е осуществляется с помощью посылки объекту-приемнику сообщения. Такая посылка представляет собой требование выполнить объектом-приемником действия, описанные в теле метода соответствующего шаблону сообщения. Выполнение сообщения происходит, вообще говоря, таким образом: получив сообщение объект-приемник ищет соответствующий метод с соответствующим именем сообщения, начиная поиск чаще всего с того класса, экземпляром которого объект является. Если объект — класс, то метод ищется среди методов класса, а если объект — экземпляр класса, то среди методов экземпляра класса, если нужный метод находится, то он выполняется и как результат его выполнения обязательно возвращается некоторый объект, который информирует того, кто послал сообщение, что выполнение его завершилось с содержащимся в возвращаемом объекте результатом.

А если нет такого метода в классе? Тогда метод ищется в ближайшем суперклассе исходного класса. При этом очень важно что ищется - метод класса или метод экземпляра. Если ищется метод экземпляра, то для поиска суперкласса используется иерархия классов, а если ищется метод класса, то используется иерархия для метаклассов. Если нужного метода нет в первом суперклассе, то в



соответствующей иерархии поиск продолжается в следующем суперклассе и так далее, пока не доберемся до класса **Object**. Если и в нем нужного метода нет, то выдается диагностическое сообщение о том, что объект-приемник не может понять и выполнить посланного ему сообщения.

Таким образом, посылка сообщения включает в себя:

- определение объекта-приемника, которому посылается сообщение;
- определение, если необходимо, объектов-аргументов сообщения;
- определение нужного метода, который ищется в классе или суперклассе;
- возвращение некоторого объекта, как результата выполнения метода в ответ на сообщение.

Итак, что произойдет в ответ на **Association key: 'Index' value: 344017**? Прежде всего класс **Association** отправится на поиски метода с селектором **key:value:**. Поскольку объект **Association** — класс, поиск будет вестись среди методов класса и начнется поиск с методов класса **Association**. Там такой метод есть и он начнет выполняться. Тело этого метода состоит из одного выражения **^(self key: aKey) value: anObject**. При сравнении с шаблоном сообщения, аргумент **aKey** станет равным строке **'Index'**, а аргумент **anObject** — целому числу **344017**. Таким образом, начнет выполняться выражение **^(self key: 'Index') value: 344017**. В начале выражения стоит символ возврата значения, следовательно, тот объект, который получится в результате вычисления и будет возвращен, как результат выполнения метода. Первая часть выражения заключена в круглые скобки **(self key: 'Index')** и выполняется первой. Первое слово в скобках — имя псевдопеременной **self**, Более подробно о ней и других псевдопеременных позже, а пока нам достаточно знать, что оно просто обозначает приемник сообщения, то есть класс **Association**. Таким образом, нам надо выполнить выражение **Association key: 'Index'**. По уже знакомой схеме, класс **Association** отправится на поиск метода с селектором сообщения **key:** и аргументом **'Index'**. Такой метод найдется среди методов класса в классе **Association** и начнет выполняться. Тело этого метода состоит из выражения **^self new key: aKey**, таким образом начнет выполняться выражение **^ Association**

new key: 'Index'. В этом выражении классу **Association** сначала посылается сообщение с селектором **new**. Соответствующего метода среди методов класса **Association** нет. Поскольку нужен метод класса, то класс **Association** начинает поиск нужного метода, используя иерархию метаклассов. Иерархия классов для класса **Association** имеет вид:

```
Object
  Magnitude
    Association
```

Тогда, в соответствии с правилами построения иерархии для метакласса класса **Association**, то есть для **Association class**, иерархия будет несколько иная:

```
Object
  Behavior
    Class
      Object class
        Magnitude class
          Association class
```

Именно эту иерархию будет использовать класс **Association**, чтобы отыскать метод соответствующий сообщению **new**. Продвигаясь по иерархии снизу вверх, нужный метод найдется в классе **Behavior** и будет выполнен, возвращая новый экземпляр класса **Association**. Теперь этот новый экземпляр является объектом, которому посылается следующее сообщение **key: 'Index'**. Поиск нужного метода начнется среди методов экземпляра в классе **Association**. Там нужный метод есть, и состоит он из одного выражения **key := anObject**, которое присваивает значение первой переменной только что созданного экземпляра класса **Association**. Поскольку в последнем методе не указан возвращаемый объект, метод возвращает приемник сообщения, то есть экземпляр класса **Association**, у которого инициализирована первая переменная. Этим завершается выполнение выражения **^ Association new key: 'Index'**, но для завершения выполнения первоначального сообщения остается завершить выполнение сообщения **^ ( Association key: 'Index') value: 344017**. Для этого надо выполнить сообщение **value: 344017**, посылаемое экземпляру класса **Association**, полученному в результате всех предыдущих вычисле-

ний. Это сообщение выполняется совершенно аналогично тому, как выполнялось вновь созданным экземпляром класса `Association` сообщение `key: 'Index'`, только теперь устанавливается значение второй переменной ассоциативной пары и экземпляр класса `Association` с обоими инициализированными переменными возвращается как результат вычисления выражения

Association key: 'Index' value: 344017

и именно на этот объект будет теперь ссылаться переменная `myIndex`.

Но почему посылаемые сообщения выполнялись именно в таком порядке? Зачем понадобились скобки и как синтаксически правильно записать выражение и метод? Об этом в следующем разделе, в котором рассмотрим формальное и более детальное описание синтаксиса языка программирования `Smalltalk`.

Что же касается вопросов практического создания нового класса или удаления существующего, создания в классе нового метода, удаления или модифицирования старого метода, то об этом позже — в разделе посвященном интерфейсу пользователя.

## 3 Синтаксис языка `Smalltalk`

### 3.1 Форма Бэкуса-Наура

Формальное описание синтаксиса языка `Smalltalk` представим с помощью расширенной формы Бэкуса-Наура (РФБН). Определение синтаксиса РФБН нуждается в предварительном описании.

Спецификация на языке РФБН является последовательностью синтаксических правил. Правая сторона каждого правила определяет синтаксис посредством имен других правил и терминальных символов языка. При этом

- последовательность символов, набранная **рубленным шрифтом**, идентифицирует терминальные символы определяемого языка;
- идентификатор — последовательность букв и цифр, начинающаяся с буквы;

- круглые скобки ( и ) объединяют вместе альтернативные термины и, следовательно, в выражении может быть только одно из них;
- вертикальные черточки | разделяют альтернативные термины; следовательно, в выражении может быть любое из них;
- квадратные скобки [ и ] идентифицируют необязательные выражения;
- фигурные скобки { и } идентифицируют выражения, которые могут появляться нуль или большее число раз.

## 3.2 Синтаксис языка и примеры

Приведем синтаксис языка **Smalltalk**, описанный посредством формул РФБН. Для большинства определений рассмотрим простые поясняющие примеры. Мы будем описывать стандарт языка, поэтому укажем в определении только буквы английского алфавита и только их будем использовать во всем остальном изложении, если не оговорено что-либо другое. Если же язык расширен буквами национальных алфавитов, эти определения надо соответствующим образом расширить.

### 3.2.1 Литералы системы

Начнем с определения букв, цифр, чисел, символов, строк, системных имен. Эти объекты языка называются литералами. Их особенность состоит в том, что для создания их как объектов систем, их достаточно записать, то есть представить в литеральной форме, что с точки зрения организации системы соответствует посылке сообщений к метаклассам, в результате которых порождаются новые экземпляры соответствующих классов.

*латинскаяПрописнаяБуква* = A | B | C | D | E | F | G | H | I | J | K |  
L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z.

*латинскаяБуква* = *латинскаяПрописнаяБуква* | a | b | c | d | e | f |  
g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z.

*буква* = *латинскаяБуква*.

*цифра* = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

*большаяЦифра* = *цифра* | *латинскаяПрописнаяБуква*.

*цифры* = *цифра* [ *цифра* ].

*большиеЦифры* = *большаяЦифра* *большаяЦифра* .

*целое число* = [ *цифры r* ] [ - ] *большиеЦифры*

*рациональная дробь* = *целое число* / *целое число*

*число* = *целое число* | *рациональное число* | [ *цифры r* ] [ - ]  
*большиеЦифры* [ . *большиеЦифры* ]  
[ e [ - ] *цифры* ] |

Примеры чисел:

16r2FA1. Число перед 'r' (всегда в десятичной форме) указывает на основание системы счисления числа. Это пример целого числа — экземпляра класса **SmallInteger** — в 16-тиричной системе счисления. Если 'r' перед числом отсутствует, значит данное число записано в десятичной системе счисления.

12, -23, 234, 8r2712, 16r-FF. Примеры различных целых чисел.

34/55, 8r-5366/8r5F1, 2r10011101/2r1011. Это рациональные дроби — экземпляры класса **Fraction**.

3.1416, -3.1459. Примеры десятичных рациональных чисел — экземпляров класса **Float**. Поскольку 'r' отсутствует, то основание системы счисления равно 10.

2.54e-2. Это экземпляр класса **Float**, записанный в экспоненциальной форме и представляющий число  $2.54 * 10^{-2} = 2.54/100 = 0.0254$ .

16rFe-2 =  $15 * 16^{-2}$ , 8r-2e-3 =  $-2 * 8^{-3}$ . Экземпляры класса **Float** в экспоненциальной форме в системе счисления, отличной от десятичной, и их десятичные представление. Как видно из примеров, символ **e** в экспоненциальной форме записи рационального числа всегда совпадает по значению с основанием системы счисления. Обратите внимание еще и на то, что показатель степени числа **e** всегда задается в десятичной форме.

Теперь определения символов, строк, комментариев, системных имен:

*символСообщения* = , | + | / | " | \* | ~ | < | > | = | @ | % | — | & |  
? | !

$\text{символ} = \text{символСообщения} \mid \text{буква} \mid \text{цифра} \mid [ \mid ] \mid - \mid " \mid ( \mid ) \mid ^ \mid$   
 $; \mid \$ \mid \# \mid :$   
 $\text{символьнаяКонстанта} = \$\text{символ} \mid \$' \mid \$"$   
 $\text{строка} = ' \{ \text{символ} \mid ' \mid " \} '$   
 $\text{комментарий} = " \{ \text{символ} \mid ' \} "$   
 $\text{системноеИмя} = \# \{ \text{символ} \mid ' \mid " \}$

Вот простые примеры, построенные по этим определениям:

символьные константы (экземпляры класса `Character`):

`$A, $~, $*, $f, $7, $,;`

строки (экземпляры класса `String`):

`'Bold', 'seed', 'O" key';`

системные Имена (экземпляры класса `Symbol`):

`#Red, #Green, #at:put, #size, #HomeBook;`

комментарий:

`"Этот метод возвращает приемник сообщения".`

Отметим, что комментарий игнорируется в любом месте, за исключением случая, когда он находится внутри строки.

Чтобы завершить разговор о литералах и точно определить, что же это такое, осталось сделать только один шаг.

$\text{массив} = ( \{ \text{число} \mid \text{строка} \mid \text{системноеИмя} \mid \text{массив} \mid$   
 $\text{символьнаяКонстанта} \} )$

$\text{константаМассив} = \# \text{массив}$

$\text{литерал} = \text{число} \mid \text{строка} \mid \text{символьнаяКонстанта} \mid$   
 $\text{системноеИмя} \mid \text{константаМассив}$

Итак, литералы представляют собой конструкции языка в вычислении которых и заключается порождение новых объектов — экземпляров соответствующих классов. Следовательно, запись литерала система трактует так:

`2` — порождается экземпляр класса `SmallInteger` со значением `2`;

`'the'` — порождается экземпляр класса `String` со значением `'the'`;

`$s` — порождается экземпляр класса `Character` со значением `s`;

`#Red` — порождается экземпляр класса `Symbol` со значением `Red`.

`#(1 2)`  — порождается экземпляр класса `Array` с двумя заданными элементами `1` и `2`.

### 3.2.2 Основные синтаксические единицы системы

Следующие определения — основные понятия, которые затем используются при построении выражений, сообщений, методов. Начнем с наиболее простых.

*идентификатор* = буква { буква | цифра }  
*имяПеременной* = идентификатор  
*имяСообщения* = унарноеИмя | бинарноеИмя |  
                  ключевоеСлово { ключевоеСлово }  
*унарноеИмя* = идентификатор  
*бинарноеИмя* = - | символСообщения[символСообщения]  
*ключевоеСлово* = идентификатор:  
*номерПримитива* = <primitive: число>

Прежде чем приводить примеры, поясняющие некоторые из введенных определений, скажем, что унарноеИмя часто называют унарным селектором, бинарноеИмя — бинарным селектором, а имяСообщения, состоящее из одного или нескольких ключевых слов — ключевым селектором. Эти термины мы будем тоже использовать.

унарное имя: size, truncate, open;

бинарное имя: = , + , - , <= , >= ;

ключевое слово: at:, new:, nextAll;

номерПримитива: <примитив: 123>, <примитив: 27>.

О примитивах (или, другими словами, о примитивных методах) мы поговорим подробнее в конце этого раздела, здесь же только отметим, что примитив в тексте методов системы не определяется, а только вызывается. Создается примитив вне системы, а затем в нее встраивается.

Еще несколько “кирпичиков” языка Smalltalk, необходимых при определении синтаксических конструкций языка:

*временныеПеременные* = | имяПеременной |.  
*блок* = [ { :имяПеременной } | ] рядВыражений ]  
*первичное* = имяПеременной | литерал | блок | ( выражение )  
*унарноеСообщение* = унарноеИмя.  
*бинарноеСообщение* =  
                  бинарноеИмя ( унарноеВыражение | первичное ).

*ключевоеСообщение = ключевоеСлово  
(бинарноеВыражение | первичное)  
{ключевоеСлово (бинарноеВыражение | первичное)}.*

Приведем примеры сообщений:

унарное сообщение : `class` ;

бинарное сообщение: `* 4.3` ;

ключевое сообщение: `at: index put: anObject`.

Прежде чем двигаться дальше, рассмотрим подробнее очень важную конструкцию языка, названную блоком. Блоки — это экземпляры класса `Context`, часто используемые в системе, например, в управляющих структурах. Блок представляет отложенную последовательность действий, которые описываются выражениями языка, находящимися внутри блока. Все выражения блока заключаются в квадратные скобки и разделяются точками. Блок может иметь аргументы, которые стоят в начале блока и отделяются от выражений блока символом `|`. Каждому имени аргумента блока предшествует `:`. Блоки могут быть вложенными. Когда блок выполняется, то, если не предписано что-то другое, результатом выполнения блока является значение последнего вычисленного в блоке выражения. Блок может содержать выражение, перед которым стоит знак `^`. Значение, полученное при вычислении этого выражения и возвращается как результат вычисления блока, что приводит к завершению выполнения блока. Примеры блоков:

```
[^ true].  
[:letter | letter isVowel].  
[:a :b | a < b].
```

Блок без аргументов выполняется при посылке ему унарного сообщения `value`. Например, `[2+3] value`. Будет возвращено число 5. Блок с одним аргументом выполняется при посылке ему ключевого сообщения `value: anObject`. Например,

```
[:array | array at: 1] value: #(A B C D).
```

Возвращается символ A.



Блок с двумя аргументами выполняется при посылке ему ключевого сообщения `value: anObject1 value: anObject2`. Например, следующий блок в результате вычисления возвращает `false`:

`[ :a :b | a < b ] value: 5 value: 3.`

Разобравшись с блоками, можем продолжить определение синтаксических конструкций языка:

*унарноеВыражение* = *первичное унарноеСообщение*  
                  { *унарноеСообщение* }  
*бинарноеВыражение* = (*унарноеВыражение* | *первичное*)  
                  *бинарноеСообщение* { *бинарноеСообщение* }  
*ключевоеВыражение* = (*бинарноеВыражение* | *первичное*)  
                  *ключевоеСообщение*  
*выражениеСообщение* = *унарноеВыражение* |  
                  *бинарноеВыражение* | *ключевоеВыражение*  
*каскадноеСообщение* = *унарноеСообщение* | *бинарноеСообщение* |  
                  *ключевоеСообщение*  
*каскадСообщений* = *каскадноеСообщение* { ; *каскадноеСообщение* }  
*выражение* = { *имяПеременной* := }  
                  (*первичное* | *выражениеСообщение* { ; *каскадноеСообщение* } )  
*рядВыражений* = { *выражение* . } [ [ ^ ] *выражение* ]

Сделаем несколько замечаний относительно выражений. Отметим, что унарные выражения не имеют аргументов. Последовательность нескольких унарных выражений вычисляется слева направо. Бинарные выражения также вычисляются слева направо. В Smalltalk'е все арифметические операции имеют одинаковый приоритет, поэтому при выполнении выражения `2 + 4 * 7` получим число 42. Для изменения порядка вычисления в выражениях необходимо использовать круглые скобки. Так например, вычисляя `2 + (4 * 7)` получим число 30. Теперь примеры выражений:

- унарные выражения:  
10 factorial.  
letter isVowel not.  
objectA class name size.

- бинарные выражения:

$x @ y$ .

$(\text{object1 size}) + (\text{object2 size})$ .

$2 + 4 * 7$ .

- ключевые выражения:

`anArray at: 2`.

`Rectangle origin: point1 corner: point2`.

`Prompter prompt: 'Name:' default: ''`.

- выражения:

`array1 := Array new: 3`.

`array1 at: 1 put: 'Smitt'`.

`'letter' size + 5`.

При выполнении выражений, в которых присутствуют все виды селекторов, работает правило: унарные селекторы имеют самый высокий приоритет, затем идут бинарные селекторы и, наконец, ключевые. Для задания иного порядка вычисления надо использовать круглые скобки. Рассмотрим пример смешанного выражения:

`arrayA at: arrayA size — 1 put: anObject class`.

Это выражение будет вычисляться следующим образом:

- сообщение `size` посылается объекту, определенному переменной `arrayA`;
- сообщение `- 1` посылается результату сообщения `size`;
- сообщение `class` посылается переменной `anObject`;
- сообщение `at:put:` посылается переменной `arrayA` с аргументами, являющимися результатами сообщений `'-'` и `class`.

Чтобы сразу был ясен порядок вычисления, разбираемое выражение можно записать, используя скобки:

`arrayA at: ((arrayA size) — 1) put: (anObject class)`.

Отметим еще один важный момент: одному и тому же объекту можно послать сразу несколько сообщений (каскад сообщений), разделяя посылаемые сообщения друг от друга символом ”;”. Например,

```
arrayA at: 1 put: 'one';  
      at: 2 put: 'two';  
      at: 3 put: 'three'.
```

Продолжая описание синтаксиса языка, приведем два последних определения:

*образецСообщения = унарноеИмя | бинарноеИмя имяПеременной |  
ключевоеСлово имяПеременной { ключевоеСлово  
имяПеременной }.*  
*метод = образецСообщения  
[временныеПеременные]  
[номерПримитива]  
рядВыражений.*

Некоторые примеры методов нам уже встречались. Вернитесь на несколько страниц назад и вновь прочитайте определение класса *Association*. Наверное кое-что прояснилось и стало понятнее. Но для полной ясности надо разобраться с еще одной особенностью системы — многообразием ее переменных.

## 3.3 Переменные в языке Smalltalk

### 3.3.1 Виды переменных

Поговорим подробнее о переменных в языке Smalltalk. Их шесть видов:

- глобальные переменные системы;
- пулы (общие переменные нескольких классов);
- переменные класса;
- переменные экземпляра класса;

- временные переменные;
- псевдопеременные.

Все переменные в системе **Smalltalk**, в отличие от объектов не имеют типа. Переменные — это указатели объектов и присвоение переменной некоторого значения создает новый указатель на объект (новый псевдоним объекта), а не создает копию объекта. Для копирования объектов в системе существуют специальные методы.

Переменные различаются временем своего существования и областью действия (областью видимости). Глобальные переменные системы доступны любому объекту системы для чтения и записи и находятся все они, как уже отмечалось, в пуле с именем **Smalltalk**. Переменные содержащиеся в словарях, называемых пулами, доступны всем экземплярам классов, в которых пул объявлен при определении класса, а также всем экземплярам их подклассов. Имя переменной из пула и ее значение связываются в объект, являющийся экземпляром класса **Association**, и именно этот объект помещается в пул. Все переменные класса неявно регистрируются в пуле класса и доступны этому классу, его подклассам и всем экземплярам этого класса и его подклассов. Все эти переменные 'живут' в системе до тех пор, пока их явно не удалят. Имена всех этих переменных, так же как и имена пулов, являясь общими, начинаются с прописной буквы. Система следит за неукоснительным выполнением этого правила.

Переменные экземпляра класса хранятся в памяти внутри каждого экземпляра и исчезают вместе с экземпляром. Их имена пишутся со строчной буквы. Ссылки на эти переменные возможны только внутри того экземпляра, которому они принадлежат. На переменные экземпляра объекта-приемника некоторого сообщения в теле соответствующего метода можно ссылаться по имени этой переменной или по индексу, если класс содержит индексированные переменные. Посмотрите вновь на определение класса **Association**: в определенных в нем методах экземпляра используются переменные экземпляра **key** и **value**.

Временные переменные определяются внутри методов и блоков, они создаются в момент вызова метода или блока и уничтожаются по окончании их работы. Эти переменные имеют имена, начинающиеся (как и имена переменных экземпляра) со строчной буквы. С переменными блока мы встречались в примерах, приведенных чуть ранее при описании блоков.

В системе Smaltalk имеются еще так называемые псевдопеременные, их имена начинаются со строчной буквы, но они, тем не менее, доступны всем объектам системы. Псевдопеременные — это имена-указатели специальных объектов системы, но, в отличие от переменных, не могут быть переопределены. Псевдопеременными в системе являются: `nil`, `true`, `false`, `self`, `super`.

Псевдопеременная `nil` указывает на специальный объект — экземпляр класса `UndefinedObject`, используемый, когда необходимо указать на отсутствие какого-либо другого подходящего объекта. Псевдопеременные `true` и `false` являются единственными экземплярами классов `True` и `False` и, соответственно, обозначают логическую истину и логическую ложь.

### 3.3.2 Псевдопеременные `self` и `super`

Особое место в системе занимают псевдопеременные `self` и `super`, которые используются в теле методов и указывают на объект, который вызвал данный метод. Различаются они способом поиска метода, который необходимо выполнить объекту: `self` начинает поиск метода в классе, которому принадлежит объект-приемник, а `super` — в суперклассе того класса, в котором находится метод, содержащий псевдопеременную `super`. Использование псевдопеременной `super` не в качестве приемника (например, в качестве аргумента) полностью совпадает с использованием псевдопеременной `self`; использование `super` влияет исключительно на класс, с которого начинается поиск необходимого метода.

Чтобы разобраться в тонкостях применения этих псевдопеременных, рассмотрим простой поясняющий пример.

Проследим, как выполняются сообщения к `self`, для чего определим два класса с именами `One` и `Two` без переменных, без методов класса, причем, пусть класс `Two` — подкласс класса `One`, а класс `One` — подкласс класса `Object`.

```
Object subclass: #One
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!
```

```
!One class methods!
```

!One methods!

```
test
  ^ 1

result1
  ^ self test !!
```

One subclass: #Two  
instanceVariableNames: ''  
classVariableNames: ''  
poolDictionaries: ''!

!Two class methods!

!Two methods!

```
test
  ^ 2!!
```

Пусть `example1` — экземпляр класса `One`, а `example2` — экземпляр класса `Two`, этого можно добиться выполнив выражения

```
example1 := One new.
example2 := Two new.
```

В следующей таблице приведены результаты вычисления четырех выражений:

---

выражение	результат
<code>example1 test</code>	1
<code>example1 result1</code>	1
<code>example2 test</code>	2
<code>example2 result1</code>	2

---

Сообщение `result1` в двух приведенных выражениях вызывает один и тот же метод, определенный в классе `One`. Результат этих

выражений различен из-за сообщения к `self`, содержащегося в этом методе. Когда сообщение `result1` посылается `example1`, поиск соответствующего метода начинается с класса `One` — с класса которому принадлежит получатель сообщения. Метод для `result1` находится в этом классе, и состоит из одного выражения `^self test`. Псевдопеременная `self` ссылается на получателя сообщения, то есть на `example1`. Поиск для ответа на `test` начинается в классе, которому принадлежит получатель сообщения `test`, то есть в классе `One`. Он там есть, он выполняется и возвращает 1.

Когда же сообщение `result1` посылается объекту `example2`, поиск соответствующего метода начинается с класса `Two`. Так как нужного метода в этом классе нет, поиск продолжается в суперклассе для класса `Two` — в классе `One`. Метод для `result1` находится в этом классе, и состоит, как мы знаем, из одного выражения `^self test`. Псевдопеременная `self` ссылается на получателя сообщения, то есть на `example2`. Поиск для ответа на `test` начинается в классе, которому принадлежит получатель сообщения `test`, то есть в классе `Two`. Он там обнаруживается, выполняется и возвращает 2.

Как будут выполняться сообщения к псевдопеременной `super` объясним, введя еще два класса, с именами `Three` и `Four`, при этом, пусть класс `Four` — подкласс класса `Three`, а класс `Three` — подкласс ранее определенного класса `Two`. Пусть в классе `Four` переопределяется метод для сообщения `test`, а в классе `Three` определяются методы для двух новых сообщений — `result2` и `result3`.

```
Two subclass: #Three
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!
```

```
!Three class methods!
```

```
!Three methods!
```

```
result2
  ^self result1
```

```
result3
  ^super test!!
```

```
Three subclass: #Four
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!
```

!Four class methods!

!Four methods!

```
test
  ^ 4!!
```

Согласно определений, все экземпляры классов **One**, **Two**, **Three**, **Four** отвечают на сообщения **test** и **result1**. Определим два новых объекта, вычислив выражения

```
example3 := Three new.
example4 := Four new.
```

Попытка послать сообщение **result2** или **result3** объектам **example1** или **example2** вызовет ошибку, так как экземпляры классов **One** и **Two** не понимают эти сообщения.

Следующая таблица показывает результаты посланки шести сообщений:

---

выражение	результат
example3 test	2
example4 result1	4
example3 result2	2
example4 result2	4
example3 result3	2
example4 result3	2

---

Когда сообщение **test** посылается объекту **example3**, используется метод из класса **Two**, так как класс **Three** не переопределяет метод с селектором **test**. Объект **example4** отвечает на сообщение



`result1` возвращением числа 4, по той же причине, по которой объект `example2` возвращал число 2. При посылке сообщения `result2` к `example3`, поиск метода начинается с класса `Three`. Обнаруженный там метод возвращает результат выражения `self result1`. Поиск ответа на сообщение `result1` также начинается в классе `Three`, но нужный метод не находится ни в классе `Three`, ни в классе `Two`. Однако, он находится в классе `One` и возвращает результат вычисления выражения `self test`. Поиск ответа на сообщение `test` еще раз начинается в классе `Three`. На этот раз нужный метод обнаруживается в классе `Two` — суперклассе класса `Three` и возвращает число 2.

Эффект посылки сообщений к псевдопеременной `super` иллюстрируется ответами объектов `example3` и `example4` на сообщение `result3`. Будучи послано к `example3`, сообщение `result3` вызывает поиск метода в классе `Three`. Найденный там метод возвращает результат выражения `super test`. Так как сообщение `test` посылается к псевдопеременной `super`, поиск начинается в суперклассе класса `Three`, в классе `Two`. Метод для сообщения `test` из класса `Two` возвращает 2. Когда сообщение `result3` посылается объекту `example4`, все равно возвращается число 2, поскольку схема выполнения полностью идентична предыдущему, несмотря на переопределение классом `Four` метода для сообщения `test`.

Еще раз подчеркнем: использование псевдопеременной `super` не означает, что поиск нужного метода начнется в суперклассе получателя первоначального сообщения. Использование `super` означает, что поиск начинается в суперклассе того класса, в котором содержится метод, содержащий псевдопеременную `super`. В рассмотренном примере, классом, содержащим метод с псевдопеременной `super`, был класс `Three`, поэтому поиск нужного метода начинался в классе `Two`. Даже если бы класс `Three` переопределял метод для сообщения `test`, результатом вычисления выражения `example4 result3` было бы число 2.

Рассмотренные классы иллюстрируют еще одну особенность объектно-ориентированного программирования — полиморфизм. Обратите внимание, что в классах `One`, `Two`, `Four` содержался метод с одним и тем же селектором `test`, но выражения (тело метода), исполняемые объектами разных классов в ответ на это сообщение, всегда были разными. Все определялось классом, которому принадлежал приемник сообщения. Определение в подклассе метода с тем же селектором, что и в суперклассе, называют перегрузкой (или, просто, переопределением метода), а само явление, когда за одним и тем

же селектором в разных классах спрятаны разные последовательности выражений языка, называют полиморфизмом. Как следует из предыдущего, использование псевдопеременной **super** позволяет объектам получить доступ к методам суперкласса, даже если эти методы были переопределены в подклассах.

## 3.4 Примеры синтаксических конструкций языка **Smalltalk**

### 3.4.1 Методы и примитивные методы

Более внимательно рассмотрим примеры синтаксических конструкций языка, в которых используются те или иные типы переменных. По ходу дела более внимательно посмотрим на то, как устроены методы в классах системы. Начнем с метода класса из системного класса **Time**.

**millisecondsToRun: aBlock**

“Возвращает число миллисекунд, необходимых для выполнения блока **aBlock**.”

| **startTime** |

**startTime** := self millisecondClockValue.

**aBlock** value.

^ self millisecondClockValue — **startTime**

Слова **millisecondsToRun: aBlock** — это шаблон (образец) сообщения, который представлен ключевым сообщением (состоящим из одного ключевого слова **millisecondsToRun:** и аргумента **aBlock**). Сразу после шаблона сообщения в кавычках “...” находится комментарий к определяемому методу. Затем в теле метода строкой | **startTime** | вводится временная переменная метода **startTime**, с которой могут работать все выражения в теле метода. Далее в теле метода находятся три выражения, которые определяют производимые методом операции. Сначала выражение

**startTime** := self millisecondClockValue

производит присваивание значения временной переменной **startTime**. В этом выражении псевдопеременная **self**, поскольку рассматрива-

емый метод — метод класса, ссылается на класс `Time`. Сообщение `millisecondClockValue`, посылаемое к `sf self` вызывает метод класса `Time`, который вычисляет время в миллисекундах от начала суток до текущего времени (времени вызова метода); на это число и будет ссылаться переменная `startTime`. Второе выражения метода

`aBlock value`

вычисляет блок `aBlock`. Здесь `aBlock` — это имя переменной, которая является экземпляром класса `Context`, которому в теле метода посылается унарное сообщение `value`, определенное как метод экземпляра класса `Context`. Третье выражение

`^ self millisecondClockValue - startTime`

сначала по тому же правилу, что и первое выражение, вычисляет текущее время, из которого затем вычитается время `startTime`. Здесь мы встречаемся с бинарным выражением, которое состоит из бинарного имени `'-'` (минус) и переменной `startTime`. Сообщение `' - aNumber'` определено как метод экземпляра в классе `Number` и может посылаться только экземплярам этого класса, в данном выражении оно посылается числу `self millisecondClockValue`. Результат выполнения этого бинарного сообщения возвращается как результат выполнения классом `Time` определяемого метода. После завершения выполнения метода временная переменная `startTime` прекращает свое существование.

Прежде чем рассмотреть еще один метода класса из системного класса `Time`, напомним, что работа в системе осуществляется объектами, которым посылаются сообщения и которые посылают сообщения другим объектам. Фактическое выполнение действий, однако, производится с помощью примитивных методов, изначально присутствующих в системе. Примитивные методы выполняют операции низкого уровня, такие как арифметические операции, доступ к индексированным переменным экземпляра, доступ к внешним устройствам и так далее. Они также используются для реализации некоторых операций высокого уровня, критичных с точки зрения производительности. Примитивные методы идентифицируются в программе на языке `Smalltalk` целым номером примитива, заключенным в угловые скобки вместе с ключевым словом `primitive:`.

Эта конструкция должна следовать после образца сообщения. Например, в классе `Time` реализован следующий метод класса

```
clockTickPrimitive: anInteger
```

```
    “Частный — разрешить прерывание по таймеру.”
```

```
    <primitive: 18>
```

```
    ^self primitiveFailed.
```

Данный метод языка `Smalltalk` состоит из двух частей: части, написанной на ассемблере, и части, написанной на `Smalltalk`'е. Часть, написанная на ассемблере, определяется номером, следующим за ключевым словом `primitive:`. Затем следует часть, написанная на языке `Smalltalk`. Выполняется сначала часть, написанная на ассемблере. Ее выполнение заканчивается либо успехом (возвращением некоторого объекта как результата выполнения метода), либо неудачей. Если примитивный метод не может выполниться, то выполняется второе выражение, в котором объекту-приемнику исходного сообщения, в данном случае классу `Time` (именно на него ссылается псевдопеременная `self`), посылается сообщение `primitiveFailed`. Метода с таким селектором нет среди методов класса `Time`, поэтому он ищется по уже упоминавшимся правилам и находится в классе `Object`. Знак `^` перед `self` означает, что в случае ошибки примитива, результат выполнения классом `Time` сообщения `self primitiveFailed`, возвращается как результат выполнения этим классом всего метода.

Это разделение ответственности между ассемблерной и смолтовской частями работает очень эффективно, поскольку ассемблерная часть управляет наиболее часто используемыми, но простыми случаями. Программу на `Smalltalk` 'е изменять намного проще, чем на ассемблере, поэтому часть, написанная на `Smalltalk` 'е, отвечает за нечастые, но более сложные ситуации. По документации, поставляемой с языком, или непосредственно просматривая все классы системы, можно выяснить набор всех примитивных методов, используемых в системе. Мы на этом останавливаться не будем.

### 3.4.2 Пример класса с переменной класса

Чтобы посмотреть, что такое переменная класса и как ее можно использовать, рассмотрим еще один учебный пример, представляющий собой небольшую модификацию класса `FinancialHistory` из кни-

ги [1]. Построим класс с именем **HomeBook**, экземпляры которого позволят нам вести простейший учет семейных доходов и расходов. Определим этот класс как подкласс класса **Object** с четырьмя переменными экземпляра и одной переменной класса. В переменной класса с именем **TaxRate** будем хранить ставку подоходного налога, взимаемого в России с любого вида доходов, сегодня она равна 12%. Мы не ставим сейчас перед собой задачу по созданию совершенного класса и, чтобы не усложнять модель, не будем в ней учитывать многих моментов, в том числе и того факта, что в зависимости от суммы годового дохода меняется и ставка налога. Цель примера — объяснить использование переменных класса и переменных экземпляра, и по ходу дела ввести некоторые новые объекты и механизмы.

Итак, для чего надо сделать ставку налога переменной класса? Для того, чтобы все экземпляры этого класса и его подклассов, если они будут, имели к ней доступ, это во-первых, и во-вторых, такое определение позволит, в случае изменения закона о величине ставки налога, очень просто произвести необходимое изменение в нашей модели — надо будет только изменить значение этой переменной. Чтобы определить значение этой переменной класса надо иметь соответствующий метод — он должен быть методом класса! В языке **Smalltalk** принято для метода, определяющего значения переменных класса, использовать селектор **initialize**. Как видно из приводимого ниже определения класса **HomeBook**, тело этого метода состоит только из одного выражения присваивания **TaxRate := 0.12**.

Теперь опишем переменные экземпляра класса **HomeBook**.

**incomes** Эта переменная представляет собой набор ассоциативных пар (такие объекты называются словарями и являются экземплярами класса **Dictionary**. Каждая пара содержит ключ — строку, описывающую источник дохода, и значение — сумму, полученную из этого источника.

**expenditures** Это тоже словарь, в каждой ассоциативной паре которого ключ — строка, описывающая причину, по которой деньги были потрачены, а значение — потраченная сумма.

**sumIncomes** Это число, отражающее совокупный доход из всех источников.

**sumExpenditures** Это число, отражающее совокупный расход на все семейные нужды.

Теперь шаг за шагом проследим за определением класса **HomeBook** и всех его методов, как класса, так и экземпляра. Как уже отмечалось, наш класс — подкласс класса **Object**, поэтому определение выглядит следующим образом:

```
Object subclass: #HomeBook
  instanceVariableNames:
    'incomes expenditures sumIncomes sumExpenditures '
  classVariableNames: 'TaxRate'
  poolDictionaries: '' !
```

Среди методов класса, должны быть методы, которые устанавливают значения переменных класса (у нас — метод **initialize**), и методы, если они необходимы, которые создают новый экземпляр класса и должным образом инициализируют переменные созданного экземпляра (методы **new**, **withIncomes:withExpenditure:**).

**!HomeBook class methods!**

**initialize**

```
"Определить ставку подоходного налога."
TaxRate := 0.12
```

**new**

```
"Создать новый экземпляр класса и установить исходные
значения всех его переменных по умолчанию."
^super new withIncomes: 0 withExpenditure: 0
```

**withIncomes: amountIncomes withExpenditure: amountExpenditures**

```
"Создать новый экземпляр класса и установить исходные
значения его переменных sumIncomes и sumExpenditures
равными значениям соответствующих аргументов сообщения."
^super new
    withIncomes: amountIncomes
    withExpenditure: amountExpenditures
```

Отметим некоторые моменты в определяемых методах. Прежде всего, обратите внимание, что мы переопределили метод **new** из класса **Behavior** (не забывайте, что методы класса исполняются

метаклассом класса, а метаклассы имеют свою иерархию, связанную с иерархией классов через класс `Class`). Поэтому в теле метода с селектором `new` из нашего класса, чтобы избежать бесконечного цикла, связанного с повторяющимся вызовом методом `new` самого себя, стоит псевдопеременная `super`, которая при поиске метода для сообщения `new`, посланного ей, пропускает `HomeBook class` — метакласс класса `HomeBook` и потому исполняет метод с селектором `new` из класса `Class`. В методе с селектором `withIncomes:withExpenditure:` тоже используется псевдопеременная `super`, но уже для того, чтобы избежать создания экземпляра по умолчанию, а затем последующего переопределения значений переменных экземпляра.

Второй момент связан с тем, что инициализация переменных вновь созданного экземпляра производится с помощью метода экземпляра с селектором `withIncomes:withExpenditure:`, который совпадает с селектором метода класса. Но никаких коллизий при этом не происходит! Ведь сообщения с совпадающими селекторами посылаются разным объектам системы: или классу, который является единственным экземпляром своего метакласса, или экземпляру этого класса. Поскольку только принявший сообщение объект принимает решение как ему выполнить полученное сообщение, поиск соответствующего метода происходит в разных местах иерархии: класс ищет метод, начиная со своего метакласса, находит его там и выполняет, а экземпляр класса начинает поиск в самом классе, и тоже его там находит и выполняет.

Обратимся теперь к методам экземпляра класса `HomeBook`. В этих методах используются сообщения к словарям, числам и логическим псевдопеременным `true` и `false`. В ответ на сообщение `includesKey: aKey`, посланное словарю, возвращается логическая псевдопеременная `true` или `false` в зависимости от того, есть или нет в словаре-приемнике ассоциативная пара с ключом `aKey`. По сообщению `at: aKey` словарь возвращает значение ассоциативной пары с ключом `aKey`, если такая пара есть. По сообщению `at: aKey put: aValue` в словаре-приемнике изменяется значение в ассоциативной паре с ключом `aKey`, если пара с таким ключом есть в словаре, если нет — такая пара создается и добавляется в словарь. Числам посылаются сообщения иницирующие выполнение арифметических операций, а также сообщение `rounded`, по которому возвращается ближайшее к числу-приемнику целое число. И наконец, логической псевдопеременной `true` или `false` (результату выполнения сообщения к словарю `includesKey: aKey`) посылается сообщение `ifTrue:`

`aTrueBlock` `ifFalse: aFalseBlock`, которое возвращает или результат вычисления блока `aTrueBlock`, если приемник сообщения совпадает с `true`, или результат вычисления блока `aFalseBlock`, если приемник сообщения совпадает с `false`.

В методе с селектором `sumRate` используется переменная класса `TaxRate`. Тело метода выполняет вычисление суммы, выплаченной в виде подоходного налога, исходя из предположения, что в переменной экземпляра `amountIncomes` храниться сумма реально полученных денег, а из начисленной суммы вычитается только подоходный налог.

!HomeBook methods!

`withIncomes: amountIncomes withExpenditure: amountExpenditures`

"Инициализировать все переменные вновь созданного экземпляра."

`amountIncomes := amountIncomes.`

`amountExpenditures := amountExpenditures.`

`incomes := Dictionary new.`

`expenditures := Dictionary new`

`cashOnHand`

"Возвратить сумму оставшихся денег."

`^ amountIncomes — amountExpenditures`

`amountIncomes`

"Возвратить общую сумму доходов."

`^ amountIncomes`

`amountExpenditures`

"Возвратить общую сумму расходов."

`^ amountExpenditures`

`totalReceivedFrom: source`

"Возвратить сумму денег, полученную из источника `source`."

`(incomes includesKey: source)`

`ifTrue: [^ incomes at: source]`

`ifFalse: [^ 0]!`

`receive: amount from: source`



```
"Запомнить в словаре incomes сумму денег (amount),  
полученную из источника source, и увеличить сумму доходов."  
incomes at: source  
    put: (self totalReceivedFrom: source) + amount.  
amountIncomes := amountIncomes + amount!
```

```
totalSpentFor: reason  
    "Возвратить сумму денег, потраченную по причине reason."  
    (expenditures includesKey: reason)  
    ifTrue: [^expenditures at: reason]  
    ifFalse: [^0]
```

```
spend: amount for: reason  
    "Запомнить в словаре expenditures сумму денег (amount),  
    потраченную по причине reason, и увеличить сумму расходов."  
    expenditures at: reason  
        put: (self totalSpentFor: reason) + amount.  
    amountExpenditures := amountExpenditures + amount
```

```
sumRate  
    "Возвратить сумму, выплаченную в виде подоходного налога."  
    ^((amountIncomes * TaxRate) / (1 - TaxRate)) rounded
```

Перед тем как использовать экземпляры класса **HomeBook**, необходимо инициализировать переменную класса. Это произойдет, если вычислить выражение

HomeBook initialize

Теперь можно создать экземпляр класса с именем **MyHomeBook** и с его помощью вести учет доходов и расходов. Поскольку мы запишем имя экземпляра с большой буквы, то оно станет глобальной переменной системы, будет занесено в системный словарь **Smalltalk** и будет доступно всем объектам.

```
MyHomeBook := HomeBook new.  
MyHomeBook receive: 300000 from: 'salary';
```

```
        receive: 100000 from: 'forArticles';
        receive: 50000 from: 'lottery'.
MyHomeBook spend: 80000 for: 'foods';
        spend: 20000 for: 'books';
        spend: 30000 for: 'clothes'.
MyHomeBook receive: 300000 from: 'salary';
        spend: 30000 for: 'books'.
```

После того как экземпляр создан и в него внесены первые записи о доходах и расходах, можем послать ему следующие сообщения и получить такие возвращаемые значения:

выражение	результат
MyHomeBook amountIncomes	750000
MyHomeBook amountExpenditures	160000
MyHomeBook cashOnHand	590000
MyHomeBook totalReceivedFrom: 'salary'	600000
MyHomeBook totalSpentFor: 'books'	50000
MyHomeBook sumRate	102273

Из примеров, приведенных и в этом разделе и в предыдущих, становится ясно, что для успешного использования системы при написании собственных классов, надо хорошо знать те классы, которые поставляются фирмой-производителем. К подробному изучению основных системных классов мы теперь и перейдем.

# 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] Шлеер С., Меллор С., *Объектно-ориентированный анализ: моделирование мира в состояниях*. — Киев: «Диалектика», 1993
- [4] *Технологический модуль объектно-ориентированного программирования (ТМООП). Руководство пользователя*. — М.: Ин-т проблем информатики АН СССР, 1990
- [5] *Технологический модуль объектно-ориентированного программирования (ТМООП). Описание языка*. — М.: Ин-т проблем информатики АН СССР, 1990
- [6] *Технологический модуль объектно-ориентированного программирования (ТМООП). Описание применения*. — М.: Ин-т проблем информатики АН СССР, 1990
- [7] Иванов А., Кремер Ю., *Язык Smalltalk: концепция объектно-ориентированного программирования* // КомпьютерПресс — 1992, № 4 — С. 21–31
- [8] Безруков Д., Голосов А., *Система Smalltalk-80 и объектно-ориентированное программирование* // Искусственный интеллект: В 3-х кн. Кн. 3. Программные и аппаратные средства: Справочник — М.: «Радио и связь», 1990 — С. 67–71
- [9] Фути К., Судзуки Н., *Языки программирования и схемотехника СБИС*: пер. с япон. — М.: «Мир», 1988

# Содержание

<b>1</b>	<b>Объектно-ориентированный подход в программировании</b>	<b>3</b>
<b>2</b>	<b>Язык программирования Smalltalk</b>	<b>9</b>
2.1	Краткая история языка . . . . .	9
2.2	Иерархия классов Smalltalk и ее особенности . . . . .	11
2.3	Определение класса и пример класса . . . . .	18
2.4	Сообщения и методы в языке Smalltalk . . . . .	23
<b>3</b>	<b>Синтаксис языка Smalltalk</b>	<b>27</b>
3.1	Форма Бэкуса-Наура . . . . .	27
3.2	Синтаксис языка и примеры . . . . .	28
3.2.1	Литералы системы . . . . .	28
3.2.2	Основные синтаксические единицы системы . . . . .	31
3.3	Переменные в языке Smalltalk . . . . .	35
3.3.1	Виды переменных . . . . .	35
3.3.2	Псевдопеременные <code>self</code> и <code>super</code> . . . . .	37
3.4	Примеры синтаксических конструкций языка Smalltalk . . . . .	42
3.4.1	Методы и примитивные методы . . . . .	42
3.4.2	Пример класса с переменной класса . . . . .	44
	<b>Литература</b>	<b>51</b>