Смолток. Язык и его реализация

Адэль Голдберг и Дэвид Робсон

13 августа 2008 г.

Краткое оглавление

Предисловие

Успехи в конструировании и производстве компьютерной техники предоставили многим людям возможность напрямую работать с компьютерами. Нужно подобное продвижение в конструировании и производстве компьютерных программ для того чтобы взаимодействие с компьютером было продуктивным настолько насколько это возможно. Система Смолток — это результат десятилетия исследований по созданию компьютерных программ предназначенных для высокофункционального и интерактивного взаимодействия с персональным компьютером.

Данная книга это первое детальное описание системы Смолток. Она делится на четыре главные части:

Часть I — обзор идей и синтаксиса языка программирования.

Часть II — аннотированое и иллюстрированное определение функций системы.

Часть III — пример разработки и реализации небольшого приложения.

Часть IV — определение виртуальной машины Смолтока.

Первая часть вводит подход Смолтока к представлению информации и манипулированию ей. Словарь с помощью которого обсуждается Смолток состоит из пяти слов — объект, сообщение, класс, экземпляр и метод. Определяются данные термины и вводится синтаксис языка программирования Смолток.

Вторая часть книги содержит описания различных видов объектов уже присутствующих в системе программирования Смолток. Новые виды объектов могут быть добавлены программистом, но большой набор объектов уже содержится в системе. Приводятся сообщения которые можно посылать любому виду объектов, также

приведён комментарий для них и пример использования.

Третья часть книги это пример добавления в систему новых видов объектов. Она описывает добавление дискретного моделирования, моделей управляемых событиями таких как мойка автомобилей, банки или информационные системы. Некоторые читатели могут найти полезным прочесть третью часть книги сразу после чтения первой части, заглядывая в определение языка во второй части когда не ясно значение предложений Смолтока.

Четвёртая часть книги описывает как может быть реализована виртуальная машина для Смолтока. Данная виртуальная машина предоставляет объектно-ориентированное хранилище, ориентированную на сообщения обработку данных и графически ориентированное взаимодействие с пользователем. Она в основном интересна читателям которые хотят реализовать систему Смолток, или читателям которые хотят детально понимать реализацию ориентированной на сообщения системы.

Цель написания книги

Написание данной первой книги о системе Смолток было сложной задачей, в основном из за sociology of the system's creation, и также из за необходимости людей в различной информации о такой системе. Мы можем разделить различные причины сложности данной задачи на четыре категории:

- Смолток это предвосхищение.
- Смолток основан на малом количестве концепций, но определяется с помощью необычной терминологии.
- Смолток это графическая интерактивная среда.
- Смолток это большая система.

Смолток это предвосхищение

В начале 1970-х годов в Xerox Palo Alto Research Center Learning Research Group началась работа по поиску путей с помощью кото-

рых различные люди смогли бы эффективно и успешно использовать мощность компьютеров. В 1981 название группы було изменено на Software Concepts Group или SCG. Задачей SCG было создание мощной информационной системы, такой чтобы пользователь мог загружать в неё информацию, получать к ней доступ и управлять ей и чтобы система могла развиваться в соответствии с идеями пользователя. Количество и виды компонентов системы должны увеличиваться в соответствии с увеличением осведомлённости пользователя об эффективном использовании системы.

Стратегия SCG по реализации данной задачи была сконцентрирована на двух принципиальных областях исследования: язык описания (язык программирования) который служит интерфейсом между моделями в голове пользователя и моделями в оборудовании компьютера, и на языке для взаимодействия (интерфейс с пользователем) который согласовывает систему взаимодействия человека и компьютера. Исследования Смолтока следовало двух- или четырёхгодичному циклу: создать систему соответствующую текущему пониманию; реализовать приложения которые проверяют возможности системы по поддержанию данных приложений; и в конце основываясь на полученном опыте переформулировать понимание о нужных программах и перепроектировать язык программирования и/или интерфейс пользователя.

Система Смолток прошла через пять таких циклов. Исследования до сих пор продолжаются. Мы надеемся что существование детального описания результатов текущих исследований поможет сообществу работающему в соответствии с видинием SCG. Продолжающееся развитие исследований означает что программы описанные в данной книге это «движущиеся цели» и информация в данной книге представляет только остановку на длинном пути. Удержание поезда на станции достаточно долго чтобы написать о нём книгу делает задачу написания книги сложной.

Смолток основан на малом количестве концепций

Смолток основан на малом количестве концепций, но определяется с помощью необычной терминологии. Из за единообразной ориентации системы на концепцию объект-сообщение, присутствует немного новых концепций программирования которые нужно изучить для понимания Смолтока. С одной стороны, это значит что читатель может рассмотреть все концепции быстро и затем изучать различные способы с помощью которых эти концепции применяются в системе. Эти концепции представляются при помощи определения пяти слов, упомянутых ранее, которые создают словарь Смолтока — объект, сообщение, класс, экземпляр и метод. Эти пять слов определяются в терминах остальных, так что это выглядит так что читатель должен знать всё до понимания чего-либо.

Смолток это среда

Смолток это графическая, интерактивная среда программирования. Следуя видению персонального компьютера, Смолток разработан так что каждый компонент системы доступен пользователю и может быть представлен различными способами для обзора и манипулирования. Задача интерфейса с пользователем в Смолтоке — попытка создать визуальный язык для каждого объекта. Оборудование нужное для системы Смолток включает графический экран с высоким разрешением и указывающее устройство такое как графическое перо или мышка. При помощи данных устройств пользователь может выбирать информацию видимую на экране и посылать сообщения для взаимодействия с этой информацией.

Один из способов представления деталей системы Смолток это начать с интерфейса пользователя и описывать каждую возможность доступных объектов. Такое представление можно начать с сценариев по которым программист может взаимодействовать с системой. Каждый сценарий будет снимком динамической системы. В линейном, статическом способе книга будет пытаться представить динамику множества путей доступа к большому и разнообразному

количеству информации.

Данная сторона системы является важной частью того что предоставляет Смолток как среда написания программ. Однако, для объяснения способа работы графического интерфейса пользователя, читатель должен вначале понимать язык программирования. Поэтому данная книга изменяет представление системы начиная с самого языка. Информация об объектах системы которые поддерживают интерфейс пользователя отделена и, за исключением корневых классов графики, не представлена в данной книге. Другая книга по Смолтоку об интерфейсе пользователя представляет детальное обсуждение реализации данных объектов системы (Смолток: интерактивная среда разработки (Smalltalk-80: The Interactive Programming Environment by Adele Goldberg))

Смолток это большая система

Система Смолток состоит из многих компонентов. Она включает объекты которые предоставляют функции обычно относимые к операционной системе: автоматическое управление памятью, файловая система, управление монитором, редактирование текста и изображений, ввод с клавиатуры и указывающего устройства, отладчик, отслеживание производительности, планировщик исполнителя, компиляция и декомпиляция. Существует множество видов объектов о которых нужно знать.

Смолток построен по модели взаимодействующих объектов. Большое приложение рассматривается как состоящее из тех же единиц что и вся система. Взаимодействие между наиболее простыми объектами рассматривается также как и высокоуровневое взаимодействие между компьютером и пользователем. Объекты поддерживают модульность — работа любого объекта не зависит от внутренних деталей других объектов. Сложность системы преодолевается при помощи минимизации взаимозависимостей между компонентами системы. Сложность также преодолевается при помощи группирования подобных компонентов; в Смолтоке это достигается при помощи классов. Классы это главный механизм расширения Смолтока. Определённые пользователем классы становятся частью системы на тех же правах что и корневые классы системы. Подклассы поддер-

живают возможность дробить систему без повторения тех же идей в различных местах.

Управление сложностью это ключевой вклад подхода Смолтока в программировании. Первые примеры языка очень простые, взяты из типовых упражнений по программированию представленных во многих книгах по различным языкам программирования. Поэтому эти примеры очень короткие, иллюстрирующие одну или две идеи. При помощи этих примеров нельзя увидеть достоинства Смолтока. Кроме того, они могут быть выполнены на другом языке и, возможно, даже лучше. Достоинства Смолтока становятся видимыми при разработке и реализации больших приложений, или когда производится модификация самой системы. Напрмер, рассмотрим *Словарь*, часто используемую структуру данных в системе Смолток. Можно разработать, реализовать, отладить и установить новое представление словарей без нарушения работы запущенной системы. Это возможно до тех пор пока интерфейс сообщений при помощи которого происходит взаимодействие с другими системами не нарушается.

Система Смолток поддерживает множество интересных инструментов разработки, в особенности классы и экземпляры как единицы организации и разделения информации, и подклассы как способ наследования и улучшения существующих возможностей. Вместе с интерактивным способом процесса создания программы, система Смолток предоставляет мощное средство для макетирования приложений и улучшения существующих.

Написание книги о такой мощной системе предполагает что некоторые вещи опущены. Повторим что в данной книге не описываются детали интерфейса с программистом и способа которым создаются графические приложения. Мы фокусируемся на языке и корневых классах системы.

Зачем читать эту книгу

Эта книга предполагает от читателя некоторое количество компьютерной граммотностьи. Предполагается что читатель

- знает почему системы программ это хорошая идея;

- это программист или разработчик языков программирования который знает хорошо хотя бы один язык;
- знаком с идеей синтаксиса выражений и выполнения выражений интерпретатором;
- знаком с набором инструкций компьютера, управляющими структурами такими как итерация и рекурсия и знаком с ролью структур данных;
- заинтересован в улучшении контроля над представлением и манипуляцией информации в компьютерной системе;
- ищет новые идеи создания систем программ (приложений) которые поддерживают возможность выражать программные решения способом близким к натуральному представлению решения.

Часть данной книги предназначена для программистов интересующихся как реализовать язык и его среду разработки на данном типе оборудования. Из за наличия различных архитектур задаче «переносимости» придаётся особое значение. Переносимость означает что для реализации работающей системы толька малая часть функций должна действиетльно быть реализована при помощи оборудования. Данная книга даёт пример достижения такой переносимости.

Список участников

Система Смолток основывана как на идеях языка Симула (Simula) и на идеях Алана Кэя (Alan Kay), который первым предложил нам попробовать создать однородную объектно-ориентированную систему. Текущее воплощение этих идей является результатом двух связанных усилий: исследования, выполненного в Xerox Palo Alto Research Center; и сотрудничества с группой по рецензированию результатов исследований.

В августе 1980-го года несколько производителей оборудования были приглашены на обсуждение нашей второй попытки написать

книгу о системе Смолток и о её последней реализации. Первая попытка описывала систему Смолток-76 и была оставлена в надежде создать более переносимую систему для распространения вне центра исследований Xerox. Второй попыткой была книгой частично исторической, частично посвящённой изложению нашей точки зрения на основные концепции персонального компьютера, частично, описывала функциональные особенности новой системы Смолток. Мы хотели назвать её "Smalltalk Dreams and Schemes"("Мечты и схемы Смолтока"), как отражение её двойственной цели. Производители, которые терпеливо рецензировали наши материалы, работали в Apple Computer, Digital Equipment Corporation, Hewlett-Packard и Tektronix. Эти компании были выбраны потому, что они разрабатывали компьютерную технику. Мы надеялись что просматривая наши материалы они изучат наши необычные задачи и посветят некоторое время проблеме создания оборудования специально для Смолток-подобных систем. Мы знали что оборудование, имевшиеся на рынке в то время, и даже планируемое к выпуску в будущем, имело недостаточную мощность для поддержки наших идей. Вместо того чтобы разрабатывать программы под уже готовое оборудование мы решили попробовать получить оборудование разработанное с учётом требований нужных нам программ.

Производители выделили сотрудников из своих исследовательских лабораторий для чтения второй версии книги. Книга сильно выиграла от многочисленных обсуждений и интенсивной работы рецензентов. Часть книги была полностью переписана в результате их конструктивной критики. Рецензенты ответственны за то, что мы продолжали работать над завершением книги, но они не несут ответственности за ошибки и неточности. Каждая из групп рецензентов по крайней мере один раз реализовала систему, чтобы проверить наши спецификации виртуальной машины Смолтока. Текущая спецификация отражает их тщательное рецензирование.

Как авторы этой книги, мы несём ответственность за создание описания системы Смолток. Но честь создания системы принадлежит всем членам Software Concepts Group. Всем этим людям мы выражаем нашу признательность, благодарность и любовь. Дэн Ингалс (Dan Ingalls) руководил проектированием и разработкой системы. Peter Deutsch на Dorado, Glenn Krasner на Alto и Kim McCall

на Dolphin (также известном как Xerox 1100 Scientific Information Processor) приобретали опыт реализации виртуальной машины на компьютерах фирмы Xerox. Идеи интерфейса с пользователем и их реализации, и управление процессом выпуска версий выполнили: James Althoff (разработка интерфейса с пользователем), Robert Flegal (проектирование графического редактора), Ted Kaehler (работа над проблемами виртуальной памяти), Diana Merry (our text guru) и Steve Putz (управление версиями). Peggy Asprey, Marc Meyer, Bill Finzer и Laura Gould, стараясь сохранить программы в постоянно развивающейся системе, проверяли важные изменения системы. Плодотворное чтение рукописи в разные периоды создания книги выполнили Michael Rutenberg, Michael Madsen, Susanne Bodker и Jay Trow. Помощь в редактирование оказали Rachel Rutherford и Janet Moreland.

Глава 18 о ядре графики Смолтока взята из рукописи подготовленной Дэном Ингалсом для журнала Byte; глава 30 была написана Larry Tesler. Рисунки из глав 18,19 и 20 создны Robert Flegal (рисунки 18.1 и 20.1), Дэном Ингалсом и Адель Голдберг (рисунки 20.2 и 20.3). Steve Putz предложил ценную помощь в создании изображений для главы 17. Изображения перед первой и второй частями и все изображения, располагаемые в начале глав с 1 по 20, созданы Адель Голдберг. Изображения перед третьей и четвёртой частью и все изображения расположенные в начале глав с 21 по 30, были созданы Robert Flegal. Эти изображения созданы при помощи редактора графики системы Смолток в комбинации со сканером изображения с низким разрешением разработанным Joseph Maleson.

Мы также выражаем наши благодарности всем участникам рецензирования. С ними мы установили постоянные научные связи которые, надеемся, будут развиваться и расти. Мы ощущали постоянную поддержку от администратора лаборатории Bert Sutherland. Рецензентами и реализаторами были: из Apple, Rick Meyers и David Casseres; из Digital Equipment Corporation Stoney Ballard, Eric Osman, Steve Shirron; из Hewlett-Packard Alec Dara-Abrams, Joe Falcone, Jim Stinger, Bert Speelpenning и Jefi Eastman; и из Tektronix Paul McCullough, Allen Wirfs-Brock, D. Jason Penney, Larry Katz, Robert Reed и Rick Samco. Мы благодарим их компании и администраторов за терпение и готовность уйти от промышленных стандартов, по крайней мере на

один краткий миг: в Apple Steve Jobs и Bruce Daniels; в Digital Larry Samburg; в Hewlett-Packard Paul Stoft, Jim Duley и Ted Laliotis; и в Tektronix Jack Grimes и George Rhine. Сотрудники Tektronix подготовили детальные обзоры на аудиоленте, так что мы могли не только видеть наши ошибки, но и слышать их!

Надеемся что эта книга и её компаньоны будут способствовать распространению языка Смолток в компьютерном сообществе. Если это произойдет то этот успех будет разделён с нашими коллегами из Xerox Palo Alto Research Center.

Постскрипт о печати этой книги

Текст оригинала этой книги был представлен издателю на магнитной ленте. Лента содержала коды форматирования определяющие различные виды объектов текста книги. Конечный формат каждого типа объекта определялся издателем. Этот процесс прошёл гладко в значительной степени благодаря усилиям и терпению Eileen Colahan из International Computaprint Corporation и Fran Fulton, нашего редактора, и также благодаря сотрудничеству с Sue Zorn, Marshall Henrichs и Jim DeWolf из Addison-Wesley.

Многие рисунки представляющие в книге примеры экрана системы Смолток и все художественные заставки к частям и главам были напечатаны с помощью системы Platemaker, разработанной Gary Starkweather и Imaging Sciences Laboratory of PARC. Мы хотели бы поблагодарить Gary, Eric Larson, и Julian Orr за возможность доступа к Platemaker.

Адэль Голдберг, Дэвид Робсон
Пало Альто, Калифорния
Январь, 1983

Оглавление

Часть I

Обзор идей и синтаксиса языка

В первой части данной книги представлен обзор языка Смолток. Первая глава вводит основные концепции и словарь языка программирования Смолток без введения его синтаксиса. В этой главе описываются объекты, сообщения, классы, экземпляры и методы. Эти понятия обсуждаются более подробно в следующих четырёх главах вместе с описанием синтаксиса языка. Вторая глава описывает синтаксис предложений. Предложения позволяют определять сообщения и ссылаться на объекты. Третья глава описывает синтаксис классов и методов. Классы и методы позволяют создавать новые типы объектов и изменять существующие объекты. Последние две главы описывают две важные роли классов в системе Смолток. Эти роли позволяют создавать подклассы и метаклассы.

Глава 1

Объекты и сообщения

Оглавление

Объекты представляют компоненты системы Смолток. Например, объекты представляют:

- числа
- цепи знаков
- очереди
- словари
- прямоугольники

- папки с файлами
- текстовые редакторы
- программы
- компиляторы
- вычислительные процессы
- финансовые истории
- отображение информации

Объект состоит из некоторой собственной памяти и набора операций. Природа операций объекта зависит от типа представляемого компонента. Объекты представляющие числа вычисляют арифметические функции. Объекты представляющие структуры данных сохраняют и выдают информацию. Объекты представляющие позиции и площади выдают информацию о своём положении и площади.

Сообщение это запрос к объекту на выполнение его операции. Сообщение определяет нужную операцию, но не как данная операция должна быть выполнена. Получатель, объект к которому было послано сообщение, определяет как выполнить запрошенную операцию. Например, сумма выполняется путём посылки сообщения объекту представляющему число. Сообщение определяет что требуется выполнить сложение и также определяет какое число нужно добавить к получателю. Сообщение не определят как должна быть выполнена операция. Как должны быть выполнено сложение определяет получатель. Вычисления это важная возможность объектов которая может быть единообразна выполнена с помощью посылки сообщений.

Набор сообщений на который может отвечать объект называется его интерфейсом с остальной системой. Единственный способ взаимодействовать с объектом это его интерфейс. Ключевым свойством объекта является то что его собственная память может быть доступна только при помощи операций. Ключевым свойством сообщений является то что они являются единственным способом выполнения операций объекта. Данные свойства гарантируют что реализация

одного объекта не зависит от внутренних деталей другого объекта, только от сообщений на которые он отвечает.

Сообщения гарантируют модульность системы т.к. они определяют тип нужных операций, но не способ которым эти операции должны выполняться. Например, в системе Смолток существует несколько представлений численных значений. *Дроби*, *Малые целые*, *Большие целые* и числа с плавающей точкой представляются различными способами. Они все понимают одни и те же сообщения запрашивающие вычисление их суммы с другим числом, но каждое представление предполагает различные способы вычисления данной суммы. Чтобы взаимодействовать с числом или другим объектом нужно только знать на какое сообщение он отвечает, но не то как он его выполняет.

Другие среды программирования также используют объекты и сообщения для поддержания модульной разработки. Например, Симула использует их для описания моделирования и Гидра использует их для описания возможностей операционной системы в распределённой системе. В системе Смолток сообщения и объекты используются для реализации всей системы программирования. Как только будут поняты сообщения и объекты то сразу станет доступна вся система.

Пример широко используемой структуры данных в программировании это словарь, который связывает имена и значения. В системе Смолток словарь представлен при помощи объекта который выполняет две операции: связывает имя со значением и находит последнюю ассоциацию к данному имени. Программист пользующийся словарём должен знать как выполнять данные операции при помощи сообщений. Объект словарь понимает сообщения которые отвечают на подобные запросы «связать имя Brett со значением 3» и «какое значение связано с именем Dave?» Т.к. всё объекты имена Brett или Dave и значения 3 или 30 представлены объектами. Хотя любопытный программист может захотеть узнать как связь представлена внутри словаря, знание этого внутреннего представления не требуется для успешного использования словаря. Знание реализации словаря нужно только программисту который работает над определением самого объекта словаря.

Важная часть проектирования программы на Смолтоке это опре-

деление типов объектов которые нужно описать и на какие сообщения они должны отвечать для взаимодействия между этими объектами. Когда программист определяет сообщения которые могут быть посланы объекту он определяет язык. Конечно, подходящий выбор объектов зависят от цели с которыми создаётся объект и детальность манипулирования информацией. Например, если создаётся модель парка развлечений для сбора информации об очередях на разные аттракционы, то будет полезно описать объекты представляющие аттракционы, работников которые управляют ими, очереди и людей посещающих парк. Если цель модели включает слежение за расходом пищи в парке, то нужны объекты представляющие эти расходуемые ресурсы. Если происходит слежение за количеством денег потраченных в парке, то нужно представить в системе детали о цене аттракционов.

Выбор объектов это первый главный шаг в проектировании программы на Смолтоке. Нельзя сказать ничего определённого о «правильном способе» выбора объектов. Как и в любом процессе проектирования это умение приобретается с опытом. Различные выборы влекут различные базы для расширения программы или для использования объектов для других целей. Опытные программисты на Смолтоке знают что объекты созданные для программы могут быть более полезными для других программ если для этих объектов определён семантически полный набор функций. Например, словарь чьи ассоциации могут быть добавлены и удалены более полезен чем версия словаря только с возможностью добавления ассоциаций.

1.1 Классы и экземпляры

Класс описывает реализацию набора объектов которые представляют один и тот же вид компонентов системы. Отдельные объекты описываемые классом называются экземплярами. Класс описывает для своих экземпляров собственную память и способ выполнения операций. Например, в системе существует класс описывающий реализацию объектов представляющих прямоугольные области. Этот класс описывает как отдельные экземпляры запоминают положение их области и также как экземпляры выполняют операции которые предоставляет прямоугольная область. В системе Смолток каждый

объект это экземпляр класса. Даже объекты которые представляют уникальные компоненты системы реализованы как единственные экземпляры класса. Программа на Смолтоке состоит из создания классов, создания экземпляров этих классов и определения последовательности обмена сообщениями между этими объектами.

Экземпляры класса похожи и в способе доступа к их внешним и внутренним свойствам. Внешние свойства объекта это сообщения которые предоставляет его интерфейс. Все экземпляры класса имеют один и тот же интерфейс т.к. они представляют один и тот же вид компонента. Внутренние свойства объекта это набор переменных экземпляра которые составляют его собственную память и набор методов которые описывает способ выполнения операций. Переменные экземпляра и методы не доступны напрямую для других объектов. Все экземпляры класса используют один и тот же набор методов для описания своих операций. Например, все экземпляры которые представляют прямоугольник отвечают на один и тот же набор сообщений и все используют один и тот же метод для ответа на сообщение. Каждый экземпляр имеет свои переменные экземпляра, но все они вместе одинаковое количество переменных экземпляра. Например, все экземпляры которые представляют прямоугольники, имеют две переменных экземпляра.

У каждого класса есть имя которое описывает тип компонента представляемого экземплярами. Имя класса будет записываться при помощи специального шрифта т.к. это часть языка программирования. Класс чьи экземпляры представляют последовательность знаков называется *Цепь*. Класс чьи экземпляры представляю точки в пространстве называется *Точка*. Класс чьи экземпляры представляют прямоугольные области называется *Прямоугольник*. Класс чьи экземпляры представляют процесс вычисления называются *Процесс*.

Каждая переменная экземпляра это ссылка на собственную память объекта, называемую её значением. Значения двух переменных экземпляра *Прямоугольника* это экземпляры *Точки* которые обозначают противоположные углы прямоугольной области. Тот факт что Прямоугольник имеет две переменных экземпляра, или что эти переменные ссылаются на *Точки* это строго внутренняя информация, недоступная вне данного *Прямоугольника*.

Каждый метод класса описывает как выполнять операцию запрошенную данным типом сообщения. Когда данный тип сообщения посылается к экземпляру класса выполняется метод. Методы используемые всеми *Прямоугольниками* описывают как выполнять их операции в терминах двух *Точек* представляющих противоположные углы. Например, одно сообщение спрашивает *Прямоугольник* о положении его середины. Соответствующий метод описывает как вычислить центр при помощи нахождения точки лежащей посередине между противоположными углами.

В классе есть метод для каждого типа операций которые могут выполнять его экземпляры. Метод может определять некоторые изменения собственной памяти объекта и/или посылать некоторые другие сообщения. Метод также определяет объект который будет возвращён как значение вызванного сообщения. Методы объекта имеют доступ к собственным переменным объекта, но не к переменным другого объекта. Например, метод *Прямоугольника* используемый для вычисления его центра имеет доступ к двум *Точкам* на которые ссылаются его переменные экземпляра; однако, метод не может получить доступ к переменным экземпляра данных *Точек*. Метод определяет сообщения которые будут посланы *Точкам* и которые запрашивают выполнение требуемых вычислений.

Небольшой набор методов в системе Смолток не выражаются на языке Смолток. Они вызывают примитивные методы. Примитивные методы реализуются виртуальной машиной и не могут быть изменены программистом. Они вызываются с помощью сообщений точно так же как и другие методы. Примитивные методы позволяют получить доступ к оборудованию и виртуальной машине. Например, экземпляры *Целого* используют примитивные методы для ответа на сообщение $+$. Другие примитивные методы производят взаимодействие с диском и терминалом.

1.2 Пример программы

Примеры — это важная часть описания языка программирования и его среды. Большинство примеров в данной книге взяты из классов, находящихся в стандартной системе Смолток. Другие примеры взяты из классов, которые могут быть добавлены к системе

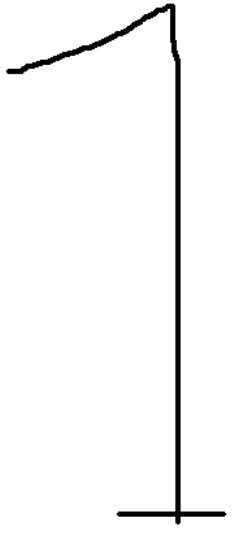


Рис. 1.1

для расширения её функциональности. Первая часть книги использует примеры из программы, которая может быть добавлена к системе для поддержания простой финансовой истории для индивидуального использования, домохозяйства или малого бизнеса. Полная программа предоставляет возможность ввода информации о финансовых операциях и вывода этой информации на экран. Рисунок 1.1 показывает как эта информация может выглядеть на экране. Две верхние части вида показывают два вида количества денег потраченных на различные нужды. Следующий вид внизу показывает как изменялось со временем количество наличных денег в течении выполнения финансовых операций.

Внизу картинки показаны две области в которых пользователь может набрать сумму для добавления нового расхода денег и добавления денег на счёт. При добавлении новой информации все три вида автоматически обновляются. На рисунке 1.2 добавлена новая

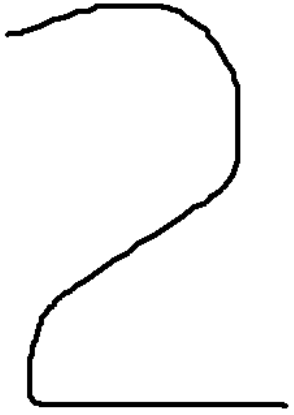


Рис. 1.2

трата на еду.

Данная программа добавляет к системе несколько классов. Эти новые классы описывают различные виды и также соответствующую информацию о финансовой истории. Класс в действительности описывающий финансовую информацию называется *Финансовая история* и будет использоваться в качестве примера в следующих четырёх главах. Данный пример программы будет использовать несколько классов уже присутствующих в системе, он будет использовать числа для представления количества денег цепи для представления причин трат и источников поступления денег.

Финансовая история используется для введения основных концепций языка программирования Смолток т.к. её функциональность и реализацию легко писать. Функциональность класса можно описать при помощи перечисления доступных операций в интерфейсе. Финансовая история предоставляет шесть операций:

1. Создание нового объекта финансовой истории с данным количеством доступных денег.

2. Запоминание, что данное количество денег было потрачено на конкретную причину.

3. Запоминание, что данное количество денег было получено из конкретного источника.

4. Нахождение количества доступных денег.

5. Нахождение количества денег, потраченных на конкретную причину.

6. Нахождение количества денег, полученных из конкретного источника.

Как описываются класса будет рассказано в главах 3, 4 и 5.

1.3 Классы системы

Система Смолток содержит набор классов, который предоставляет стандартную функциональность языка программирования и его среды: арифметику, структуры данных, управляющие структуры и возможности ввода/вывода. Функциональность этих классов будет детально описана во второй части данной книги. Рисунок 1.3 это диаграмма классов системы представленных во второй части. Вокруг связанных классов обведена рамка, группы пронумерованы для указания глав в которых можно найти описание классов.

1.3.1 Арифметика

Система Смолток содержит объекты представляющие и реальные и рациональные числа. Реальные числа могут быть представлены с точность около шести десятичных цифр. Целые с абсолютным значением меньше, чем 2^{524288} могут быть представлены точно. Рациональные числа представляются при помощи данных целых. Так же есть классы для представления линейных величина (таких как дата и время) и генератора случайных чисел.

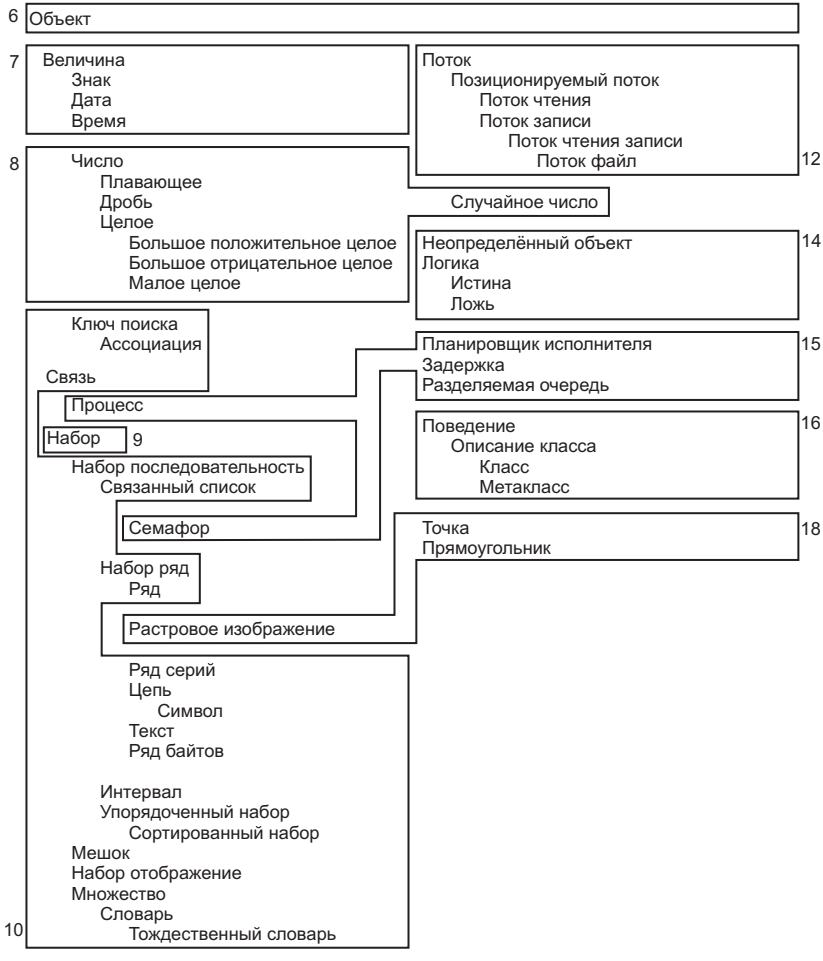

6
Объект
7
Величина
Знак
Дата
Время
8
Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое
Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл
12
Случайное число
Неопределённый объект
Логика
Истина
Ложь
14
Ключ поиска
Ассоциация
Связь
Процесс
Набор
9
Набор последовательность
Связанный список
Семафор
Набор ряд
Ряд
Растровое изображение
Ряд серий
Цепь
Символ
Текст
Ряд байтов
Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
10
Тождественный словарь
Планировщик исполнителя
Задержка
Разделяемая очередь
15
Поведение
Описание класса
Класс
Метакласс
16
Точка
Прямоугольник
18

Рис. 1.3

1.3.2 Структуры данных

Большинство объектов системы Смолток работают как некоторый вид структур данных. Однако из за того что большинство объектов также имеют другую функциональность существует набор классов представляющий более или менее структуры данных. Эти классы представляют различные типы наборов. Элементы некоторых наборов не упорядочены, а элементы других упорядочены. Из наборов с не упорядоченными элементами представлены *Мешок*, который позволяет дублировать элементы и *Множество* которое не позволяет дублирования. Также представлены *Словари*, которые связывают пары объектов. Из наборов с упорядоченными элементами для некоторых порядок элементов задаётся внешне при его добавлении, а для некоторых порядок определяется на основании свойств самого элемента. Общие структуры данных *Ряды* и *Цепи* представлены классами которые имеют ассоциативное поведение (связывают номера с элементами) и внешнее упорядочивание.

1.3.3 Управляющие конструкции

Система Смолток включает объекты и сообщения которые реализуют стандартные управляющие конструкции присутствующие в большинстве языков программирования. Они предоставляют условный выбор подобно конструкции Алгола if-then-else и условное повторение подобно конструкциям while и until. Также есть объекты предоставляющие независимые процессы и механизмы для планирования и синхронизации процессов. Для поддержания этих структур предоставляются два класса. Логические значения представляют два значения истинности а блоки представляют последовательность действий. Логические значения и блоки также используется для создания новых видов управляющих конструкций.

1.3.4 Среда программирования

В системе Смолток есть несколько классов которые помогают в процессе программирования. Это классы представляющие исходную форму (для человека) и откомпилированную форму (для выполнения машиной) методов. Объекты представляющие разборщики, ком-

пиляторы и декомпиляторы, переводящие данные две формы методов. Объекты представляющие классы, которые соединяют методы с объектами, их использующими (экземпляры классов).

Объекты, представляющие структуры, организующие методы и классы для помощи программисту в обзоре системы, и объекты, представляющие историю модификации программы, помогающие взаимодействию нескольких программистов. Даже состояние выполнения метода представляется при помощи объектов. Этот объект называется контекстом и аналогичен стеку или записи активации в других системах.

1.3.5 Просмотр и взаимодействие

Система Смолток включает классы объектов которые используются для просмотра и редактирования информации. Классы для представления графической информации представлены точками, линиями, прямоугольниками и окружностями. Т.к. система Смолток рассчитана на использование графического дисплея, то есть классы для представления и манипулирования картинками. Также есть классы для представления и манипулирования более специфическими картинками для шрифтов, текстов и курсоров.

Из данных объектов строятся другие объекты представляющие прямоугольные окна, меню и т.д. Так же есть объекты представляющие действия пользователя при помощи устройств ввода и связь между этими действиями и показываемой информацией. Классы представляющие специальные механизмы для просмотра и редактирования построенные из данных компонентов это просмотрщики классов, контекстов и документов содержащих текст и графику. Просмотрщики классов предоставляют основной механизм для взаимодействия с программами системы. Редакторы и просмотрщики системы Смолток рассмотрены в отдельной книге.

1.3.6 Связь

Система Смолток позволяет взаимодействовать с внешней информацией. Стандартная внешняя среда информации это файловая

система диска. Объекты представляют файлы и папки. Если доступна сеть, то она тоже доступна через объекты.

1.4 Сводка терминов

объект — компонент системы Смолток который представляет некоторую собственную память и набор операций.
сообщение — запрос к объекту на выполнение его операции.
получатель — объект которому послано сообщение.
интерфейс — сообщения на которые объект может отвечать.
класс — описание группы подобных объектов.
экземпляр — один из объектов описываемых классом.
переменная экземпляра — часть собственной памяти объекта.
метод — описания способа выполнения одной из операций объекта.
примитивный метод — операция напрямую выполняемая виртуальной машиной Смолтока.
Финансовая история — имя класса используемого в качестве примера в этой книге.
Классы системы — набор классов который поставляется вместе с системой Смолток.

Глава 2

Синтаксис предложения

Оглавление

Глава 1 вводит фундаментальные концепции системы Смолток. Компоненты системы представляются объектами. Объекты это экземпляры классов. Объекты взаимодействуют посредством посылки сообщений. Сообщения вызывают методы для выполнения. Данная глава вводит синтаксис предложения для описания объектов и сообщений. Следующая глава вводит синтаксис для описания классов и методов.

Выражение это последовательность символов которая описывает объект называемый значением выражения. Синтаксис представленный в данной главе объясняет какие последовательности символов образуют разрешённые последовательности. В языке Смолток есть четыре типа выражений.

1. Литералы описывают некоторые объекты константы, такие как числа и цепи знаков.

2. Имена переменных описывают доступные переменные. Значение имени переменной это текущее значение переменной с данным именем.

3. Предложение сообщение описывает сообщение к получателю. Значение сообщения определяется методом вызываемым сообщением. Этот метод находится в классе получателя.

4. Предложение блок описывает объект представляющий отложенные действия. *Блоки* используются для реализации управляющих конструкций.

Предложения встречаются в двух местах, в методах и в тексте выводимом на экран. Когда посылается сообщение выбирается метод из класса получателя и его предложения выполняются. Интерфейс пользователя позволяет выделить предложение на экране и выполнить его. Детальное описание способов выделения и выполнения предложений на экране выходит за рамки данной книги, т.к. это часть интерфейса с пользователем. Однако некоторые примеры приведены в главе 17.

Из четырёх типов выражений представленных выше только имена переменных зависят от контекста. Положение выражения в систе-

ме определяет какие последовательности букв являются допустимыми для имён переменных. Набор имён переменных доступный в предложениях метода зависит от класса в котором находится метод. Например, методы в классе *Прямоугольник* и методы в классе *Точка* имеют доступ к различным наборам имён переменных. Переменные доступные методам класса будут полностью описаны в Главах 3, 4 и 5. Имена переменных доступные для предложений на экране зависят от того где предложение выведено на экран. Все остальные стороны синтаксиса предложений не зависят от положения предложения.

Синтаксис выражений собран в диаграмме которая помещена на задней обложке данной книги. Оставшаяся часть данной главы описывает четыре типа предложений.

2.1 Литералы

Пять типов объектов могут быть обозначаться выражениями литералами. Т.к. значение выражения литерала всегда один и тот же объект, эти выражения также называются литералами константами. Пять типов литералов констант включают:

1. числа
2. индивидуальные знаки
3. цепочки знаков
4. символы
5. ряды других литералов констант

2.1.1 *Числа*

Числа это объекты которые представляют численные значения и отвечают на сообщения которые вычисляют математические выражения. Литеральное представление числа это последовательность цифр, которая может предваряться знаком минус, или/и следующей десятичной точкой и другой последовательностью цифр. Например:

```
3
30,45
-3
0,005
-14,0
13772
```

Литералы числа также могут быть представлены в недесятичной системе счисления путём предшествующих цифр в приставке основания. Приставка основания включает значение числа основания (всегда записывается в десятичной системе) и последующую букву «о». Следующие примеры определяют числа в восьмеричной системе счисление с соответствующими им десятичными значениями.

восьмеричные	**десятичные**
8o377	255
8o153	107
8o34,1	28,125
-8o37	-31

Когда основание больше десяти, тогда для цифр больших девяти используются заглавные буквы начиная с «А». Следующие примеры определяют числа в шестнадцатеричной системе счисления вмести с соответствующими десятичными числами.

шестнадцатиричные	**десятичные**
16o106	262
16oEE	255
16oAB,ГВ	172,859
-16o1,B	-1,75

Числа также могут быть записаны в научной форме с цифрами обозначающими экспоненциальный суффикс. Экспоненциальный суффикс включает букву «э» с последующим значением экспоненты (записанном в десятичной системе). Число записанное перед суффиксом умножается на основание возведённое в степень экспоненты.

научная запись	десятичное значение
1,586э5	158600,0
1,586э-3	0,001586
8о3э2	192
2о11э6	192

2.1.2 *Знаки*

Знаки это объекты которые представляют символы алфавита. Выражение литерала знака содержит знак доллара и следующий знак, например,

```
$a
$M
$-
$$
$1
```

2.1.3 *Цепи*

Цепи это объекты которые представляют последовательности знаков. Цепи отвечают на сообщения которые дают доступ к отдельным знакам, заменяют подцепи и производят сравнения с другими цепями. Литеральное представление цепи это последовательность знаков отделённая одинарными кавычками, например:

```
‘hi’
‘food’
‘the Smalltalk-80 system’
```

Любой знак может быть включён в литерал. Если одинарная кавычка включается в цепочку, то чтобы не спутать её с отделителем нужно записать кавычку дважды. Например литерал цепочка `’can’’t’` ссылается на цепочку из пяти знаков `$c`, `$a`, `$n`, `$’` и `$t`.

2.1.4 *Символы*

Символы это объекты которые представляют собой цепи используемые в системе для именования. Литеральное представление символа это последовательность букв и цифр с предшествующим знаком #, например:

```
#bill
#M63
```

Никогда не будет два символа с одинаковыми знаками, каждый символ уникален. Это позволяет сравнивать символы эффективно.

2.1.5 *Ряды*

Ряд это простая структура данных на чьё содержание можно ссылаться с помощью целых номеров от единицы до числа которое является размером ряда. *Ряды* отвечают на сообщения запрашивающие доступ к их содержимому. Литеральное представление ряда это последовательность других литералов — чисел, знаков, цепочек, символов и рядов — отделённая скобками и предшествующим знаком #. Другие литералы разделяются пробелом. *Ряд* из трёх чисел описывается выражением:

```
#( 1 2 3 )
```

Ряд из семи цепочек описывается выражением:

```
#('food' 'utilities' 'rent' 'household' 'transportation' 'taxes' 'recreation')
```

Ряд из двух рядов и двух чисел описывается выражением:

```
#( #( 'one' 1 ) #( 'not' 'negative' ) 0 -1 )
```

И ряд из числа, строки, знака, символа и другого ряда описывается выражением:

```
#( 9 'nine' $9 nine #( 0 'zero' $0 #( ) 'e' $f 'g' $h 'i' ) )
```

2.2 Переменные

Память доступная объекту состоит из переменных. Большинство из этих переменных имеет имя. Каждая переменная запоми-

нает один объект и имя переменной может быть использовано как выражение ссылающиеся на этот объект. Объекты к которым можно получить доступ в данном месте определяются тем какие имена переменных доступны. Например, контекст переменных экземпляра объекта недоступен для других объектов из за того что эти переменные могут быть использованы только в методах класса объекта.

Имена переменных это простой идентификатор, последовательность букв, цифр и неразрывных пробелов начинающаяся с буквы. Некоторые примеры имён переменных:

номер
начальный номер
редактор текста
Домашние финансы
Прямоугольник
Источник поступления

В системе существует два вида переменных, отличающихся частотой доступа к ним. Индивидуальные переменные доступные только одному объекту. Переменные экземпляра индивидуальны. Разделяемые переменные могут быть доступны более чем одному объекту. Имена индивидуальных переменных должны начинаться с маленькой буквы, имена разделяемых переменных должны начинаться с большой буквы. Первые три примера идентификаторов показных выше ссылаются на индивидуальные переменные а последние три ссылаются на разделяемые переменные.

2.2.1 Присваивания

Константа литерал всегда ссылается на один и тот же объект, но имя переменной в разное время может ссылаться на различные объекты. Объект на который ссылается переменная изменяется во время выполнения выражения присваивания. Присваивание не было упомянуто ранее как один из типов выражений из за того что любое выражение может стать присваиванием если к нему добавить приставку присваивание.

Приставка присваивание состоит из имени переменной чьё значение будет изменяться и стрелки влево. Следующий пример это

выражение литерал с приставкой присваиванием. Оно показывает что переменная *количество* должна сейчас ссылаться на объект представляемый числом 19.

количество ← 19

Следующий пример это выражение переменная с приставкой присваивания. Он показывает что переменная *номер* должна ссылаться на тот же объект что и переменная *начальный номер*.

номер ← начальный номер

Другой пример присваивания:

имя главы ← 'Синтаксис выражения'.
flavors ← #('vanilla' 'chocolate' 'butter pecan' 'garlic').

Можно добавлять более чем одну приставку присваивание, это показывает что будет изменено значение более чем одной переменной.

номер ← начальный номер ← 1.

Это выражение показывает что обе переменные *номер* и *начальный номер* должны ссылаться на число 1. Выражение сообщение и выражение блок могут также иметь приставку присваивание, как будет видно в следующих разделах.

2.2.2 Имена псевдо переменные

Имена псевдо переменные это идентификаторы которые ссылаются на объект. С этой стороны они подобны именам переменных. Имя псевдо переменной отлично от имени переменной в том что её значение невозможно изменить при помощи выражения присваивания. Некоторые из псевдо переменных это константы, они всегда ссылаются на один и тот же объект. Три важные псевдо переменные это *пусто*, *истина* и *ложь*.

пусто — ссылается на объект используемый как значение переменной когда нет другого подходящего объекта. Переменные которые не были инициализированы ссылаются на *пусто*.

истина — ссылается на объект который представляет логическую истинность. Она используется как положительный ответ на сообще-

ние с вопросом требующим подтверждения да-нет.

ложь — ссылается на объект который представляет логическую ложность. Она используется как отрицательный ответ на сообщение с вопросом требующим подтверждения да–нет.

Объекты именуемые *истина* и *ложь* называются Логическими объектами. Они представляют ответ на вопрос требующий ответа да–нет. Например, число должно отвечать *истина* или *ложь* на сообщение спрашивающие больше ли число чем другое число. Логические объекты отвечают на сообщения которые вычисляют логические функции и выполняют условные управляющие конструкции.

В системе существую другие псевдо переменные (например *сам* и *супер*) чьи значения зависят от того где они используются. Это будет описано в следующих трёх главах.

2.3 Сообщения

Сообщения представляют взаимодействие между компонентами системы Смолток. Сообщение запрашивает операцию для части получателя. Некоторые примеры выражений сообщений и взаимодействий которые они представляют:

Сообщения к числам представляют арифметические операции.
3 + 4 — вычисляет сумму трёх и четырёх.
номер + 1 — добавляет единицу к числу с именем *номер*.
номер > предел — спрашивает является ли число с именем *номер* больше чем число с именем *предел*.
тета син — вычисляет синус числа с именем *тета*.
количество корень — вычисляет корень положительного числа с именем *количество*.

Сообщения к линейным структурам данных представляют добавление или удаление информации.
список добавить первым: новый компонент — добавляет объект именуемый *новый компонент* как первый элемент линейной структуры данных с именем *список*.
список удалить последний — удалить и возвратить последний элемент из *списка*.

Сообщения к ассоциативным структурам данных (таким как *Словари*) представляют добавление или удаление информации.

возрасты от: 'Brett Jorgensen' пом: 3 — связывает цепь 'Brett Jorgensen' со значением 3 в словаре именуемом *возрасты*.

адресы от: 'Peggy Jorgensen' — ищет в словаре именуемом *адресы* объект связанный с цепью 'Peggy Jorgensen'.

Сообщения к прямоугольникам представляют графические запросы и вычисления.

рамка центр — возвращает центр прямоугольника именуемого *рамка*.

рамка содержит точку: положение курсора — отвечает истина если позиция именуемая *положение курсора* находится внутри прямоугольника именуемого *рамка*, и ложь иначе.

рамка пересечь: область вырезания — вычисляет прямоугольник который представляет пересечение двух прямоугольников именуемых *рамка* и *область пересечения*.

Сообщения к записям финансовой истории представляют транзакции и запросы

Домашнее хозяйство потратить: 32,50 на: 'коммунальные услуги' — указывает финансовой истории именуемой *Домашнее хозяйство* что 32 доллара 50 центов были потрачены на оплату коммунальных услуг.

Домашнее хозяйство общие траты на: 'еда' — спрашивает у *Домашнего хозяйства* сколько денег было потрачено на еду.

2.3.1 Селекторы и аргументы

Выражение сообщение описывает получателя, селектор и, возможно, несколько аргументов. Получатель и аргументы описываются другими выражениями. Селектор задаётся литерально.

Селектор сообщения это имя для типа взаимодействия которое описывает посылающий. Например, в следующем сообщении:

```
тета син.
```

получатель это число на которое ссылается переменная с именем *тета* и селектор это *син*. Как отвечать на сообщение определяет получатель (в данном случае, как вычислить функцию синус для своего значения). В двух предложениях сообщениях

```
3 + 4.
всего + увеличение.
```

селектор это +. Оба сообщения просят получателя вычислить и возвратить сумму. Каждое из этих сообщений содержит объект в добавок к селектору. (4 в первом выражении и *увеличение* во втором). Дополнительный объект в сообщении это аргумент который определяет количество которое нужно добавить.

Следующие два сообщения описывают один и тот же тип операции. Получатель это экземпляр *Финансовой истории* и он должен вернуть количество денег потраченных на соответсвующую статью. Аргумент показывает статью расходов. Первое предложение запрашивает количество потраченное на коммунальные услуги.

```
Домашнее хозяйство общие траты на: 'коммунаольные услуги'.
```

Количество денег потраченное на еду может быть найдено если послать сообщение с тем же селектором, но с другим аргументом.

```
Домашнее хозяйство общие траты на: 'еда'.
```

Селектор сообщения определяет какую из операций получателя нужно выполнить. Аргументы это другие объекты которые вовлечены в данную операцию.

Унарные сообщения Сообщения без аргументов называются унарными сообщениями. Например, количество доступных денег в *Домашнем хозяйстве* это значение унарного выражения сообщения

```
Домашнее хозяйство количество наличных.
```

Это сообщение называется унарным потому что используется только один объект, получатель. Селектором унарного сообщения может быть любой идентификатор. Другие примеры унарных сообщений:

```
тета син.
количество корень.
цепь имя размер.
```

Сообщения с ключевыми словами

Общий вид сообщения с одним или более аргументами это сообщение с ключевыми словами. Селектор сообщения с ключевыми словами состоит из одного или более ключевых слов, по одному перед каждым аргументом. Ключевое слово это просто идентификатор с завершающим двоеточием. Примеры выражений сообщений с одним ключевым словом:

```
Домашнее хозяйство общие траты на: 'коммунальнные услуги'.
номер макс: предел.
```

Сообщению с двумя аргументами нужен селектор с двумя ключевыми словами. Примеры выражений с двойными ключевыми слоами

```
Домашнее хозяйство потратить: 30,45 на: 'еду'.
возрасты от: 'Brett Jorgensen' пом: 3.
```

Когда ссылаются на селектор с многими ключевыми словами они соединяются. Селекторы последних двух выражений это потратить:на: и от:пом:. Может существовать сообщение с любым количеством ключевых слов, но большинство сообщений системы имеют меньше чем три ключевых слова.

Бинарные сообщения

Существует другой тип сообщений с одним аргументом, бинарное сообщение. Селектор бинарного сообщения состоит из одного или более знаков не букв. Бинарные сообщения в основном используются для арифметических сообщений. Примеры бинарных сообщений:

```
3 + 4.
всего - 1.
всего <= макс.
```

2.3.2 Возвращаемые значения

Система Смолток предоставляет два пути взаимодействия. Селектор и аргументы передают информацию к получателю о типе

ответа. Получатель передаёт информацию обратно возвращая объект который становится значением выражения сообщения. Если сообщение содержит приставку присваивание, объект возвращённый получателем будет новым значением на которое ссылается переменная. Например предложение:

сумма ← 3 + 4.

делает 7 новым значением переменной с именем *сумма*. Предложение:

икс ← тета син.

делает синус *теты* новым значением переменной с именем *икс*. Если значение *теты* это единица, новым значением *икса* становится 0,841471. Если значение *теты* это 1,5, новым значением *икса* становится 0,997495.

Число на которое ссылается переменная *номер* может быть увеличено на удиницу с помощью предложения:

номер ← номер + 1.

Даже если не нужно передавать информацию обратно к посылающему, получатель всегда возвращает значение выражения сообщения. Возвращение значения показывает что ответ на сообщение закончен. Некоторые сообщения нужны только для того чтобы проинформировать получателя о чём либо. Примерами могут быть сообщения к записям о финансовых операциях описанные следующими предложениями.

Домашнее хозяйство потратить: 32,50 на: 'коммунальные услуги'.
Домашнее хозяйство получить: 1000 из: 'зарплата'.

Получатель данных сообщений информирует отправителя только о том что он закончил запись транзакции. По умолчанию возвращаемое значение в таких случаях это обычно получатель. Так что результатом выражения:

пер ← Домашнее хозяйство потратить: 32,50 на: 'коммунальные услуги'.

будет присваивание переменой *пер* того же значения что хранится в переменной *Домашнее хозяйство*.

2.3.3 Лексический анализ

Все сообщения показанные прежде имеют получателя и аргументы заданные с помощью литералов или имён переменных. Когда получатель или аргумент сообщения описывается другим выражением сообщением, имеет значение как происходит лексический анализ выражения. Пример унарного сообщения описывающего получателя другого унарного сообщения:

```
1,5 тан округлить.
```

Унарные сообщения разбираются слева направо. Первое сообщение в примере это унарный селектор *тан* посланный $1,5$. Значение данного сообщения (число примерно равное $14,1014$) получает унарное сообщение *округлить* и возвращает ближайшее целое, 14. Число 14 это значение всего предложения.

Бинарные сообщения также разбираются слева направо. Пример бинарного сообщения описывающего получателя другого бинарного сообщения:

```
номер + смещение * 2.
```

Значение возвращённое *номером* из сообщения + *смещение* это получатель бинарного сообщения ** 2*.

Все бинарные селекторы имеют один и тот же приоритет, только порядок в котором они записаны имеет значение. Заметьте что это делает математические выражения на Смолтоке отличными от таких же выражений на большинстве других языков в которых умножение и деление имеет преимущество над сложением.

Можно использовать скобки для изменения порядка вычисления. Сообщение со скобками посылается раньше любого сообщения за скобками. Если предыдущий пример записать как:

```
номер + (смещение * 2).
```

то произведение будет выполнено раньше сложения.

Унарные сообщения имеют более высокий приоритет чем бинарные сообщения. Если унарное и бинарное сообщения используются вместе, то сначала будет послано унарное сообщение. В следующем примере:

```
frame width + border width * 2.
```

значение *frame width* это получатель бинарного сообщения чей селектор это + и чей аргумент это значение *border width*. В свою очередь, значение сообщения + это получатель сообщения * 2. Выражение разбирается так как будто были расставлены скобки следующим образом:

((frame width + (border width) * 2.

Скобки можно использовать чтобы послать бинарное сообщение до унарного. Выражение:

2 * тета син.

вычисляет два синус тета, в то время как предложение

(2 * тета) син.

вычисляет синус двойного тета.

Когда встречается ключевое слово в сообщении без скобок, оно образует один селектор. Из за этого сцепления не существует правила разбора слева на право для сообщения с ключевыми словами. Если сообщение с ключевыми словами используется как получатель или аргумент другого сообщения с ключевыми словами, то оно должно быть заключено в скобки. Предложение:

frame scale: factor max: 5.

описывает сообщение с двумя аргументами чей селектор это scale:max:. Предложение:

frame scale: (factor max: 5).

описывает два сообщения с ключевым словом чьи селекторы это *scale:* и *max:*. Значение выражения *factor max: 5* это аргумент для сообщения *frame:*.

Бинарное сообщение имеет более высокий приоритет чем сообщение с ключевыми словами. Когда унарные, бинарные и сообщения с ключевыми словами встречаются вместе в одном выражении без скобок, сначала посылаются унарные сообщения, затем бинарные, и последним сообщение с ключевыми словами. Пример:

bigFrame width: smallFrame width * 2.

выполняется так как если бы в нём были расставлены скобки сле-

дующим образом:

bigFrame width: ((smallFrame width) * 2).

В следующем примере унарное сообщение описывает получателя сообщения с ключевыми словами и бинарное сообщение описывает аргумент.

Упорядоченный набор новый добавить: значение * оценка.

Подводя итог правил разбора получаем:

1. унарные сообщения разбираются слева направо.
2. бинарные сообщения разбираются слева направо.
3. бинарные сообщения имеют приоритет над сообщениями с ключевыми словами.
4. унарные сообщения имеют приоритет над бинарными сообщениями.
5. сообщения заключённые в скобки имеют приоритет над унарными сообщениями.

2.3.4 Каскады

Есть одна специальная синтаксическая форма называемая каскад которая предписывает многим сообщения посылаться одному и тому же объекту. Любая последовательность сообщений может быть записана без каскадов. Однако, каскад часто упрощает выражение т.к. становится не нужным введение промежуточных переменных. Выражение сообщение каскад содержит одно описание получателя со следующими несколькими сообщениями разделёнными точкой с запятой. Например:

Упорядоченный набор новый добавить: 1; добавить: 2; добавить: 3.

Три сообщения *добавить:* посылаются результату *Упорядоченный набор новый.* Без каскада для этого бы потребовалось четыре сообщения и переменная. Например, следующие четыре предложения, разделённые точной, производят тот же результат что и сообщение с каскадом.

```
| врем |
Упорядоченный набор ← новый.
врем добавить: 1.
врем добавить: 2.
врем добавить: 3.
```

2.3.5 Потоки сообщений

Другой специальной синтаксической формой является поток сообщений. Он используется для посылки сообщения результату предыдущего сообщения с более низким приоритетом или тем же приоритетом (в случае ключевого сообщения) без использования скобок. Выражение потока сообщений состоит из полного сообщения и последующих сообщений без получателя (получателем является результат предыдущего сообщения) разделённых восклицательным знаком. Например:

```
имя селектор 1: 'арг 1'! селектор 2: 'арг 2'! селектор 3: 'арг 3'.
```

Без использования потока сообщений этот пример будет выглядеть так:

```
((имя селектор 1: 'арг 1') селектор 2: 'арг 2') селектор 3: 'арг 3'.
```

2.4 *Блоки*

Блоки это объекты используемые во многих управляющих конструкциях системы Смолток. Блок представляет отложенную последовательность действий. Выражение блок содержит последовательность предложений разделённых точками и отделяется с помощью квадратных скобок. Например:

```
[ целое ← целое + 1. ].
```

или

```
[ целое ← целое + 1. ряд от: целое пом: 0. ].
```

Когда встречается выражение блок, предложения заключённые в квадратные скобки не выполняются немедленно. Значение выражения блока это объект который может выполнить предложения

заключённые в скобки позже, когда это потребуется. Например, выражение

```
действия
   от: 'месячные выплаты'
   пом: [
      Домашнее хозяйство потратить: 650 на: 'аренда'.
      Домашнее хозяйство потратить: 7,25 на: 'газета'.
     Домашнее хозяйство потратить: 225,50 на: 'уплата за машину'.
].
```

не посылает сообщения *потратить:на: Домашнему хозяйству*. Оно просто связывает блок с цепью *'месячные выплаты'*.

Последовательность выражений описываемая блоком будет выполнена когда блок получит унарное сообщение *значение*. Например, следующие два выражения имеют одинаковый эффект.

```
целое ← целое + 1.
```

и

```
[ целое ← целое + 1. ] значение.
```

Объект на который ссылается выражение:

```
действия от: 'месячные выплаты'
```

это блок содержащий три сообщения *потратить:на:*. Выполнение выражения:

```
(действия от: 'месячные выплаты') значение.
```

приводит к тому что эти три сообщения посылаются *Домашнему хозяйству*.

Блок также можно присвоить переменной. Так если выполнить предложение

```
увеличивающий блок ← [ номер < номер + 1. ].
```

тогда предложение:

```
увеличивающий блок значение.
```

увеличит номер.

Объект возвращаемый сообщением *значение* посланным блоку это значение последнего предложения в блоке. Так если выполнить

предложение:

доб блок ← [номер + 1.].

то другим способом увеличения номера будет предложение:

номер ← доб блок значение.

Блок который не содержит ни одного предложения возвращает *пусто* когда ему посылается сообщение *значение*. Предложение:

[ ] значение.

имеет значение пусто.

2.4.1 Управляющие конструкции

Управляющие конструкции определяют порядок некоторых действий. Основной управляющей конструкцией в языке Смолток является последовательное выполнение. Многие непоследовательные управляющие конструкции могут быть реализованы с помощь блоков. Эти управляющие конструкции выполняются с помощью посылки сообщений блокам или сообщений с одним или несколькими блоками в качестве параметров. Ответ на такое сообщение управляющую конструкцию определяет порядок выполнения в соответствии со значениями сообщений посланных блокам.

Рассмотрение выполнения следующих выражений даёт пример способа работы блоков.

```
блок увеличить ← [ номер ← номер + 1. ].
блок сумма ← [ сумма + (номер * номер). ].
сумма ← 0.
номер ← 1.
сумма ← блок сумма значение.
блок увеличить значение.
сумма ← блок сумма значение.
```

15 действий выполняющихся в результате выполнения последовательности данных предложений это:

1. присвоить блок переменной *блок увеличить*

2. присвоить блок переменной *блок сумма*

3. присвоить число 0 переменной *сумма*
4. присвоить число 1 переменной *номер*
5. послать сообщение значение блоку *блок сумма*
6. послать сообщение * 1 числу 1
7. послать сообщение + 1 числу 0
8. присвоить число 1 переменной *сумма*
9. послать сообщение *значение* блоку *блок увеличить*
10. послать сообщение + 1 числу 1
11. присвоить число 2 переменной *номер*
12. послать сообщение *значение* блоку *блок сумма*
13. послать сообщение * 2 числу 2
14. послать сообщение + 4 числу 1
15. присвоить число 5 переменной *сумма*

Пример управляющей конструкции реализованной с помощью блоков это простое повторение, представленное в виде сообщения к *Целому* с селектором *раз повторить:* и блоком в качестве аргумента. Целое число должно отвечать за посылку сообщения *значение* блоку параметру количество раз равное самому целому. Например, следующее выражение удваивает значение переменной *количество* четыре раза.

```
4 раз повторить: [ количество ← количество + количество. ].
```

2.4.2 Условные конструкции

Две наиболее часто используемые управляющие конструкции построенные с помощью блоков это условный выбор и условное повторение. Условный выбор подобен конструкции if-then-else в Алголоподобных языках а условное повторение подобно выражениям

while и until. Эти условные управляющие конструкции используют два *Логических* объекта именуемых *истина* и *ложь* которые были описаны в разделе псевдо переменные. Логические значения возвращаются сообщениями которые отвечают на вопрос да-нет (например, сообщения сравнения величин <, =, <=, >, >=, ~ =).

Условный выбор. Условный выбор действий обеспечивается с помощью сообщения *истина:ложь:* посылаемого Логическому значению и двух блоков параметров. Единственные объекты которые понимают сообщение *истина:ложь:* это *истина* и *ложь*. Они имеют противоположные результаты: истина посылает сообщение *значение* первому агрументу блоку и игнорирует второй; *ложь* посылает сообщение *значение* второму аргументу блоку и игнорирует первый. Например, следующее выражение присваивает 0 или 1 переменной *чётность* в зависимости от того делится ли на 2 значение переменной *число*. Бинарное сообщение \\ вычисляет функцию деления по модулю.

```
число \\ 2 = 0 истина: [ чётность ← 0. ] ложь: [ чётность ← 1. ].
```

Значение возвращаемое сообщением *истина:ложь:* это значение блока который был выполнен. Предыдущий пример можно также записать так:

```
чётность ← число \\ 2 = 0 истина: [ 0. ] ложь: [ 1. ].
```

В дополнение к сообщениям *истина:ложь:* существуют два сообщения с одним ключевым словом которые определяют только одну альтернативу. Селекторы этих сообщений это *истина:* и *ложь:*. Эти сообщение имеют тот же эффект что и сообщение *истина:ложь:* когда один из аргументов это пустой блок. Например, данные предложения имеют один и тот же эффект:

```
номер <= предел истина: [ итог ← итог + (список от: номер). ].
```

и

```
номер <= предел истина: [ итог ← итог + (список от: номер). ] ложь:
[ ].
```

Т.к. значение пустого блока это *пусто*, следующее выражение должно присваивать *пусто последнему элементу* если *номер* больше *предела*.

последний элемент ← номер > предел истина: [список от: номер.].

Условное повторение. Условное повторение действий обеспечивается с помощью сообщения блоку *пока истина:* и другого блока в качестве аргумента. Блок получатель посылает себе сообщение *значение* и если ответ это *истина* он посылает параметру сообщение *значение* и начинает с начала снова посылая себе сообщение *значение*. Когда ответ получателя на сообщение значение становится *ложь* он заканчивает повторение и возвращается из метода *пока истина:*. Например, условное выполнение может быть использовано для инициализации всех элементов ряда именуемого *список*.

```
номер ← 1.
[ номер <= список размер. ]
  пока истина: [ список от: номер пом: 0. номер ← номер + 1. ].
```

Блоки также понимают сообщение с селектором *пока ложь:* которое повторяет выполнение аргумента пока значение получателя ложно. Так что следующее выражение эквивалентно предыдущему.

```
номер ← 1.
[ номер > список размер. ]
  пока ложь: [ список от: номер пом: 0. номер ← номер + 1. ].
```

Программист может выбирать соответствующее сообщение которое делает назначение повторения более ясным. Значение возвращаемое сообщениями *пока ислина:* и *пока ложь:* всегда *пусто*.

2.4.3 Аргументы блока

Чтобы сделать более простым написание некоторых непоследовательных управляющих конструкций блоки могут иметь один или более аргументов. Аргументы блока определяются с помощью добавления в начало блока идентификаторов с двоеточием в начале имени отделённых от выражений блока вертикальной чертой. Следующие два примера описывают блоки с одним аргументом.

```
[ :ряд | всего ← всего + ряд размер. ].
```

и

```
[ :новый элемент | номер ← номер + 1. список от: номер пом: новый элемент. ].
```

Наиболее часто блоки с аргументами используются чтобы реализовать функции которые применяются ко всем элементам структуры данных. Например, многие объекты предоставляющие различные структуры данных отвечают на сообщение *делать:*, которое берёт в качестве аргумента блок с одним параметром. Объект который получает сообщение *делать:* выполняет блок один раз для каждого элемента содержащегося в структуре данных. Каждый элемент является значением аргумента при выполнении блока параметра. Следующий пример вычисляет сумму квадратов первых пяти простых чисел. Результат это значение суммы.

```
сумма ← 0.
#( 2 3 5 7 11 ) делать: [ :прост | сумма ← сумма + (прост * прост).
].
```

Сообщение *собрать:* создаёт совокупность значений полученных от блока когда он применяется к элементам получателя. Значение следующего выражения это ряд квадратов первых пяти простых чисел.

```
#( 2 3 5 7 11 ) собрать: [ :прост | прост * прост. ].
```

Объект который выполняет эту управляющую конструкцию передаёт значение блоку при помощи сообщения *значение:*. Блок с одним параметром отвечает на сообщение *значение:* путём помещения аргумента сообщения *значение:* в параметр блока и выполнения предложений блока. Например, результатом вычисления следующих предложений будет значение 7 в переменной *всего*.

```
доб размер ← [ :ряд | всего ← всего + ряд размер. ].
всего ← 0.
доб размер значение: #( $а $б $в ).
доб размер значение: #( 1 2 ).
доб размер значение: #( $г $д ).
```

Блок может принимать более чем один параметр. Например:

```
[ :а :б | а * а + (б * б). ].
```

или

```
[ :frame :clippingBox | frame intersect: clippingBox. ].
```

Блок должен иметь сколько же аргументов сколько есть ключевых слов *значение:* в селекторе метода выполняемом им. Два вышеупомянутых блока должны быть выполнены с помощью сообщения с двумя ключевыми словами *значение:значение:*. Два аргумента сообщения определяют соответственно два значения параметра блока. Если блок получает сообщение с другим количеством аргументов то будет сообщено об ошибке.

2.5 Сводка терминов

предложение — последовательность знаков которая описывает объект.
литерал — предложение описывающее константу, такую как число или цепь.
символ — цепь чья последовательность знаков гарантированно отличается от последовательности любого другого символа.
ряд — структура данных чьи элементы доступны при помощи целых чисел.
имя переменной — предложение описывающее текущее значение переменной.
присваивание — предложение описывающее изменение значения переменной.
псевдо переменная — предложение подобное имени переменной. Но, в отличии от имени переменной, значение псевдо переменной нельзя изменить при помощи присваивания.
получатель — объект который получил сообщение.
селектор сообщения — имя типа операции сообщения посланного получателю.
аргумент сообщения — объект который определяет дополнительную информацию для операции.
унарное сообщение — сообщение без аргументов.
ключевое слово — идентификатор с завершающим двоеточием.
ключевое сообщение — сообщение с одним или несколькими аргументами чей селектор состоит из одного или более ключевых слов.

бинарное сообщение — сообщение с одним аргументов чей селектор состоит из специальных знаков.
сообщение каскад — описание нескольких сообщений к одному объекту одним предложением.
блок — описание отложенной последовательности действий.
аргумент блока — параметр который должен быть задан при выполнении блока.
значение — сообщение блоку которое запрашивает выполнение набора действий который представляет блок.
значение: — ключевое слово в сообщении блоку которое передаёт блоку аргументы; данное сообщение запрашивает выполнение набора действий который представляет блок.
истина:ложь: — сообщение Логическому значению запрашивающее условный выбор.
ложь:истина: — сообщение Логическому значению запрашивающее условный выбор.
истина: — сообщение Логическому значению запрашивающее условный выбор.
ложь: — сообщение Логическому значению запрашивающее условный выбор.
пока истина: — сообщение блоку запрашивающее условное повторение.
пока ложь: — сообщение блоку запрашивающее условное повторение.
делать: — сообщение совокупности запрашивающее перебор её элементов.
собрать: — сообщение совокупности запрашивающее преобразование её элементов.

Глава 3

Классы и экземпляры

Оглавление

Объекты представляют компоненты системы Смолток —- числа, структуры данных, процессы, файлы на диске, планировщик процессов, редакторы текста, компиляторы, и приложения. Сообщения представляют взаимодействие между компонентами системы Смолток — арифметические операции, доступ к данным, управляющие структуры, создание файла, редактирование текста, компиляция и использование приложений. Сообщения делают функциональность

объекта доступной для других объектов, при этом оставляя реализацию объекта скрытой. Предыдущая глава ввела синтаксис предложения для описания объектов и сообщений, уделяя основное внимание на способах использования сообщений для доступа к функциональности объектов. Данная глава вводит синтаксис для описания методов и классов с точки зрения как данная функциональность реализуется.

Каждый объект системы Смолток это экземпляр какого либо класса. Все экземпляры классы имеют один и тот же интерфейс; класс описывает как выполнять каждую операцию доступную в интерфейсе. Каждая операция описывается в виде метода. Селектор сообщения определяет тип операции которую должен выполнить получатель, таким образом в классе существует один метод для каждого селектора присутствующего в интерфейсе. Когда объекту посылается сообщение, вызывается метода связанный с данным типом сообщения в классе получателя. Класс также описывает какая у экземпляра есть собственная память.

У каждого класса есть имя которое описывает тип компонента представленного экземплярами данного класса. Имя класса служит для двух основных целей: оно является простейшим способом ссылки на себя, и оно является способом ссылки на класс в предложениях. Т.к. класс это компонент системы Смолток он является объектом. Имя класса автоматически становится именем глобально доступной переменной. Значение данной переменной это объект представляющий класс. Т.к. имя класса это имя глобальной переменной оно должно начинаться с большой буквы.

Новые объекты создаются с помощью посылки сообщения классам. Большинство классов отвечает на унарное сообщение *новый* созданием нового экземпляра себя. Например:

```
Упорядоченная совокупность новый.
```

возвращает новый набор который является экземпляром класса системы *Упорядоченный набор*. Новый *Упорядоченный набор* всегда пуст. Некоторые классы создают экземпляр в ответ на другие сообщения. Например, класс чьи экземпляры представляют время дня это *Время*; *Время* отвечает на сообщение *сейчас* экземпляром представляющим текущее время. Класс чьи экземпляры представ-

ляют день года это *Дата*; *Дата* отвечает на сообщение *сегодня* экземпляром представляющим текущий день. Когда создаётся новый экземпляр он автоматически разделяет методы класса который получил сообщение для создания экземпляра.

Данная глава вводит два способа представления класса, один описывает функциональность экземпляра класса а другой описывает реализацию данной функциональности.

1. Описание протокола содержит список сообщений в интерфейсе экземпляра. Каждое сообщение сопровождается комментарием описывающим операции которые должен выполнить экземпляр в ответ на сообщение.

2. Описание реализации показывает как функциональность описанная в протоколе реализуется. Описание реализации даётся в форме личной памяти экземпляра и набора методов которые описывают как экземпляр выполняет свои операции.

Третий способ представления классов это интерактивное окно, называемое браузером системы. Браузер — это часть интерфейса программы и он используется в работающей системе Смолток. Описание протокола и описание реализации создано для не интерактивных документов как данная книга. Браузер будет рассмотрен подробнее в семнадцатой главе.

3.1 Описание протокола

Описание протокола это сообщения понимаемые экземпляром данного класса. Каждое сообщение записывается с комментарием о его назначении. Комментарий описывает операции которые будут выполнены при получении сообщения и какое значение будет возвращено. Комментарий описывает почему это делается, а не как операция будет выполнена. Если комментарий не даёт указаний о возвращаемом значении то предполагается что возвращаемое значение это получатель сообщения.

Например, описанием протокола для сообщений *Финансовой истории* с селектором *потратить:на:* является:

потратить: **количество** **на:** **причина**

Запоминает что данное количество денег, количество, было потрачено на причину.

Сообщения в описании протокола описываются в форме образца сообщения. Образец сообщения содержит селектор сообщения и набор имён аргументов, одно имя для каждого аргумента которое должно иметь сообщение с данным селектором. Например образец сообщения

потратить: **количество** **на:** **причина**

Обозначает сообщения описываемые каждым из следующих трёх предложений.

Домашнее хозяйство потратить: 32,50 на: 'коммунальные услуги'.
Домашнее хозяйство потратить: цена + налог на: 'еда'.
Домашнее хозяйство потратить: 100 на: обычная причина.

Имена аргументов используются в комментарии для ссылки на аргументы. Комментарий в вышеприведённом примере указывает что первый аргумент представляет количество потраченных денег и второй аргумент представляет на что эти деньги были потрачены.

3.1.1 Категории сообщений

Сообщения которые выполняют похожие операции объединяются в категории. Категории имеют имя которое обозначает общее поведение сообщений группы. Например, сообщения *Финансовой истории* объединяются в три категории с именами: запись транзакций, справки и инициализация. Эти категории предназначены для более удобного чтения протокола человеком, они не влияют на работу класса.

Ниже показано полное описание протокола для *Финансовой истории*:

Протокол *Финансовой истории*

методы экземпляра

запись транзакций

получить: количество из: источник
Запоминает что данное количество денег, количество, было получено из источника.

потратить: количество на: причина
Запоминает что данное количество денег, количество, было потрачено на причину.

справки

количество наличных
Запрашивает о текущем количестве наличных денег.

общее поступление из: источник
Запрашивает общее количество денег поступивших из источника.

общие траты на: причина
Запрашивает общее количество денег потраченных на причину.

инициализация

инициализироать баланс: количество
Начать финансовую историю с данным количеством доступных денег.

Описание протокола предоставляет достаточно информации для программиста чтобы знать как использовать экземпляр класса. Из вышеприведённого протокола мы знаем что экземпляр *Финансовой истории* должен отвечать на сообщения чьи селекторы: *получить:из:*, *потратить:на:*, *количество наличных*, *общее поступление из:*, *общие траты на:*, и *инициализировать баланс:*. Мы можем догадаться что при создании экземпляра *Финансовой истории* нужно послать ему сообщение *инициализировать баланс:* чтобы присвоить значения его переменным.

3.2 Описание реализации

Описание реализации состоит из трёх частей:

1. имя класса
2. объявление переменных доступных экземпляру
3. методы используемые экземпляром для ответа на сообщения

Далее приведено полное описание реализации для *Финансовой истории*. Методы в описании реализации разделены на те же категории что и в описании протокола. В браузере системы категории используются для ирархического доступа к описанию частей класса. Нет специальных знаков которые разделяют различные части описании реализации. Изменения в шрифте и выделении обозначают различные части. В браузере системы эти части представлены независимо и браузер предоставляет редактор для доступа к ним.

имя класса Финансовая история
имена переменных экземпляра количество наличных приход расход

методы экземпляра

запись транзакций

получить: **количество** **из:** **источник**
приход от: источник пом: (сам общее поступление из: источник) + количество.
количество наличных ← количество наличных + количество.
потратить: **количество** **на:** **причина**
| предыдущие расходы |
предыдущие расходы ← сам общие траты на: причина.
расход от: причина пом: предыдущие расходы + количество.
количество наличных ← количество наличных − количество.

справки

количество наличных
↑ количество наличных.

```
общее поступление из: источник
  ( приход содержит ключ: источник )
    истина: [↑ приход от: источник. ]
    ложь: [↑ 0. ].
общие траты на: причина
  ( расход содержит ключ: причина )
    истина: [↑ расход от: причина. ]
    ложь: [↑ 0. ].

инициализация

инициализировать баланс: количество
  количество наличных ← количество.
  приход ← Словарь новый.
  расход ← Словарь новый.
```

Данное описание реализации отличается от описания *Финансовой истории* представленного на форзаце этой книги. Вариант на форзаце содержит дополнительный раздел называемый «методы класса» которые будут объяснены в пятой главе, также там нет методов инициализации показанных здесь.

3.3 Определение переменных

Методы класса имеют доступ к пяти различным типам переменных. Эти типы различаются областью доступности (их областью видимости) и временем жизни.

Есть два типа собственных переменных доступных только объекту:

1. Переменные экземпляра — существуют всё время жизни объекта.

2. Временные переменные — существуют во время определённых действий и доступны только в течении этих действий.

Переменные экземпляра представляют текущее состояние объекта. Временные переменные представляют промежуточное состояние

нужное для выполнения некоторых действии. Временные переменные обычно связанны с выполнением метода: они создаются когда приходит сообщение и уничтожаются когда выполнение метода заканчивается.

Другие три типа переменных могут быть доступны более чем одному объекту. Они различаются по области своей доступности.

1. Переменные класса — разделяются всем экземплярами класса.

2. Глобальные переменные — разделяются всеми экземплярами всех классов (т.е. всем объектами).

3. Переменные пула — разделяются экземплярами подмножества классов системы.

Большинство разделяемых переменных системы это либо переменные класса либо глобальные переменные. Большинство глобальных переменных ссылаются на классы системы. Экземпляр *Финансовой истории* с именем *Домашнее хозяйство* использовался в нескольких примерах в предыдущих главах. Мы испльзовали *Домашнее хозяйство* как будто оно определено как глобальная переменная. Глобальные переменные используются чтобы ссылаться на объекты не являющиеся частью других объектов.

Вспомните что имена разделяемых переменных (3-5) начинаются с большой буквы, в то время как имена собственных переменных (1-2) нет. Значение разделяемой переменной не зависит от того в каком экземпляре эта переменная используется. Значения переменных экземпляра и временных переменных зависит от экземпляра использующего метод, то есть от экземпляра который получил сообщение.

3.3.1 Переменные экземпляра

Существует два типа переменных экземпляра: именованные и нумерованные. Они различаются способом объявления и способом доступа. Класс может иметь только именованные переменные, только нумерованные или оба типа.

Именованные переменные экземпляра

Описание реализации содержит множество имён для переменных экземпляра которые используют экземпляры класса. Каждый экземпляр имеет одну переменную соответствующую каждому имени переменной экземпляра. Объявление переменных в описании реализации класса *Финансовая история* определяет три имени переменных экземпляра.

- расход ссылается на словарь который связывает причины трат с потраченным количеством.

- приход ссылается на словарь который связывает поступающие суммы с количеством поступивших денег.

- количество наличных ссылается на число представляющее количество доступных денег.

Когда предложения в методах класса используют переменные с именами *приход*, *расход* или *количество наличных* эти предложения ссылаются на значения соответствующих переменных экземпляра который получил сообщение.

Когда создаётся новый экземпляр при помощи сообщения посланного классу он получает новый набор переменных экземпляра. Переменные экземпляра инициализируются в соответствии с сообщением создания экземпляра. Метод инициализации по умолчанию присваивает каждой переменной экземпляра значение пусто.

Например, чтобы предыдущие примеры сообщений к *Финансовой истории* работали нужно выполнить предложение подобное следующему:

Домашнее хозяйство ← Финансовая история новый инициализировать баланс: 350.

Финансовая история новый — создаёт новый объект чьи все три переменных ссылаются на *пусто*. Сообщение новому экземпляру *инициализировать баланс:* присваивает трём переменным экземпляра более подходящие начальные значения.

Нумерованые переменные экземпляра

Экземпляры некоторых классов могут иметь переменные экземпляра которые не доступны с помощью имени. Они называются нумерованными переменными экземпляра. Вместо ссылки на переменную с помощью имени доступ к нумерованным переменным экземпляра осуществляется при помощи сообщения с целым аргументом, называемым номером. Т.к. нумерование это форма ассоциации, то два основных сообщения для доступа к переменным имеют те же селекторы что и сообщения к словарям — *от:* и *от:пом:*. Например экземпляр *Ряда* имеет нумерованные переменные. Если *имена* это экземпляр *Ряда* то предложение:

```
имена от: 1.
```

возвратит значение своей первой нумерованной переменной. Предложение:

```
имена от: 4 пом: 'Adele'.
```

поместит цепь ‘Adele’ как значение четвёртой нумерованной переменной экземпляра объекта *имена*. Допустимые индексы пробегают значения от единицы до числа нумерованных переменных в данном экземпляре.

Если у экземпляр класса есть нумерованные переменные то его объявление переменных должно включать строку нумерованные переменные экземпляра. Например часть описания реализации класса системы *Ряд*:

```
имя класса Ряд
нумерованные переменные экземпляра
```

Каждый экземпляр класса имеющего нумерованные переменные экземпляра может иметь различное их количество. Все экземпляры *Финансовой истории* имеют три переменные экземпляра, но экземпляры *Ряда* могут иметь любое количество переменных экземпляра.

Класс чьи экземпляры имеют нумерованные переменные экземпляра может также иметь именованные переменные экземпляра. Все экземпляры такого класса должны иметь одинаковое количе-

ство именованных переменных экземпляра, но могут иметь различное количество нумерованных переменных. Например класс системы представляющий набор чьи элементы упорядочены, *Упорядоченный набор* может иметь болше места для хранения элементов чем их текущее количество. Две именованные переменные экземпляра запоминают номера первого и последнего элемента.

имя класса Упорядоченный набор
имена переменных экземпляра первый номер последний номер
нумерованные переменные экземпляра

Все экземпляры *Упорядоченного набора* должны иметь две именованные переменные, но один может иметь пять нумерованных переменных, другой 15, другой 18 и т.д.

Именованные переменные экземпляра класса *Финансовая история* являются собственными в том смысле что доступ к этим переменным контролируются экземпляром. Класс может включать или может не включать сообщения дающие прямой доступ к переменным экземпляра. Нумерованные переменные экземпляра не собственные в данном смысле, т.к. прямой доступ к значениям этих переменных доступен при помощи посылки сообщений с селекторами от: и от:пом:. Т.к. это единственный способ получить доступ к нумерованным переменным он должен быть предоставлен.

Классы с нумерованными переменными экземпляра создают новые экземпляры при помощи сообщения *новый:* вместо использования сообщения *новый*. Аргумент сообщения *новый:* указывает количество нумерованных переменных.

```
список ← Ряд новый: 10.
```

Создаёт *Ряд* из 10 элементов, каждый из которых сначала ссылается на *пусто*. Количество нумерованных переменных экземпляра можно определить с помощью сообщения размер. Ответ на сообщение *размер*

```
список размер.
```

для данного примера это целое 10.

Выполнение каждого из следующих предложений в данном порядке:

```
список ← Ряд новый: 3.
список от: 'один'.
список от: 'два'.
список от: 'три'.
```

эквивалентно одному предложению

```
список ← #( 'один' 'два' 'три' ).
```

3.3.2 Разделяемые переменные

Переменные которые разделяются более чем одним объектом объединяются в группы называемые пулами. Каждый класс имеет два или более пула чьи переменные могут быть доступны экземплярам. Один пул разделяется всеми классами и содержит глобальные переменные, этот пул называется Смолток. Каждый класс также имеет пул который доступен только его экземплярам и содержит переменные класса.

Помимо этих двух обязательных пулов классу могут быть доступны некоторые другие пулы разделяемые несколькими классами. Например, в системе существуют несколько классов которые представляют текстовую информацию; этим классам нужен доступ к кодам АСКОИ (ASCII) для знаков которые трудно представлять визуально, таким как перевод строки, табуляция или пробел. Эти числа включены как переменные в пул именуемый *TextConstants* который разделяется классами реализующими редактирование и показ текста. Если *Финансовая история* имеет переменную класса с именем *SalesTaxRate* и разделяет словарь пула с именем *FinanicalConstnts* объявление должно записываться так:

имя класса Финансовая история
имена переменных экземпляра количество наличных приход расход
имена переменных класса SalesTaxRate
разделяемые пулы Финансовые константы

SalesTaxRate это имя переменной класса так что она может быть использована в любом методе класса. С другой стороны *Финансовые константы* это имя пула, есть переменные в пуле которые могут быть использованы в предложениях.

Чтобы объявить переменную глобальной (известной для всех классов и пользователей системы) имя переменной нужно поместить как ключ в словаре Смолток. Например чтобы сделать глобальной Все истории выполните предложение:

```
Смолток от: #Все истории пом: пусто.
```

Затем используйте предложение присваивание для задания значения *Всем историям*.

3.4 Методы

Метод описывает как объект будет выполнять одну из своих операций. Метод состоит из образца и последовательности предложений отделённых точкой. Пример метода показанный ниже описывает ответ *Финансовой истории* на сообщение информирующее его о расходовании денег:

```
потратить: количество на: причина
  | предыдущие расходы |
  предыдущие расходы ← сам общие траты на: причина.
  расход от: причина пом: предыдущие расходы + количество.
  количество наличных ← количество наличных − количество.
```

Образец сообщения **потратить: количество на: причина** показанный в этом методе должен использоваться в ответ на все сообщения с селектором *потратить:на:*.

Первое предложение в теле данного метода добавляет новое количество к уже существующему количеству денег потраченных на указанную причину. Второе предложение это присваивание которое уменьшает значение переменной *количество наличных* на новое количество.

3.4.1 Имена аргументов

Обрезец сообщения был введён выше в данной главе. Образец сообщения содержит селектор сообщения и набор имён аргументов, имя для каждого из аргументов которое должен иметь метод с данным селектором. Образец сообщения совпадает с любым сообщени-

ем которое имеет то же селектор. Класс должен иметь только один метод с данным селектором в образце сообщения. Когда сообщение посылается, то метод с совпадающим образцом сообщения выбирается из класса получателя. Предложения в выбранном методе выполняются одно за другим. После выполнения всех предложений возвращается значение пославшему сообщение.

Имена аргументов в образце метода это имена переменных которые ссылаются на аргументы действительного сообщения. Если метод показанный выше вызовется при помощи предложения:

```
Домашнее хозяйство потратить: 30,45 на: 'еда'.
```

имя переменная *количество* будет ссылаться на число 30,45 и имя переменная *цель* будет ссылаться на цепь 'еда' в течении выполнения предложений метода. Если тот же метод будет вызван при помощи предложения:

```
Домашнее хозяйство потратить: цена + налог на: 'еда'.
```

цене будет послано сообщение *+ налог* и на возвращённое значение будет ссылаться переменная *количество*. Если *цена* ссылается на 100 и *налог* на 6,5, то значение *количество* будет равно 106,5.

3.4.2 Возвращаемое значение

Метод *потратить:на:* не определяет какое значение должно быть возвращено. Поэтому, по умолчанию, будет возвращён сам получатель. Когда нужно возвратить другое значение в методе записывается одно или несколько предложений возврата. Можно возвратить значение любого предложения при помощи добавления предшествующей стрелки вверх (↑). Значение переменной можно возвратить так:

```
↑ количество наличных.
```

Значение другого сообщение можно вернуть так:

```
↑ расход от: причина.
```

Объект литерал можно возвратить так:

```
↑ 0.
```

Можно вернуть даже значение предложения присваивания:

```
↑ начальный номер ← 0.
```

Сначала происходит присваивание. Затем возвращается новое значение переменной.

Пример использования предложения возврата в реализации метода *общие траты на:*:

```
общие траты на: причина
  ( расход содержит ключ: причина )
    истина: [↑ расход от: причина. ]
    ложь: [↑ 0. ].
```

Данный метод содержит одно условное предложение. Если причина есть в расходах, возвращается соответствующее значение; иначе возвращается ноль.

Многоуровневые переходы

При использовании ↑ для возвращения значения завершается выполнение первого вызова метода в котором содержится возврат. В предыдущем примере возврат происходил из блока. При этом завершилось выполнение блока и всего метода. Если требуется завершить выполнение блока или перейти на заданное количество уровней в стеке вызовов нужно использовать многоуровневые переходы.

При многоуровневом переходе управление возвращается на заданное в виде литерала количество уровней. Чтобы задать многоуровневый переход нужно перед возвращаемым выражением записать стрелку вверх, двоеточие и целый литерал который задаёт уровень перехода. Текущий уровень равен нулю, уровень который вызвал данный метод или блок имеет номер 1 и т.д. Запись ↑*: 2* читается как возвратить со второго уровня.

Многоуровневые переходы можно использовать для завершения текущей итерации цикла. Результатом выполнения следующего примера:

```
| цепь |
цепь ← ''.
1 до: 5 делать: [ :н | н = 3 истина: [ ↑: 2 пусто. ]. цепь ← цепь, н как
```

цепь.].

будет цепь '1245'. Т.к. при значении переменной *н* равном 3 блок сообщения *до:делать:* будет завершён до выполнения предожения *цепь ← цепь, н как цепь*. В этом примере нулевой уровень это вызов блока *[↑: 2 пусто.]*, первый уровень это вызов метода с селектором *истина:*, а второй уровень это вызов блока сообщения *до:делать:* поэтому будет закончено выполнения блока цикла, поэтому в *цепи* не будет содержаться цифра 3.

3.4.3 Псевдо переменная *сам*

Наряду с переменными используемыми для ссылания на аргументы метода все методы имеют доступ к псевдо переменной с именем *сам* которая ссылается на получателя сообщения. Например, в методе *потратить:на:* сообщение *общие траты на:* посылается получателю сообщения *потратить:на:*.

```
потратить: количество на: причина
  | предыдущие расходы |
  предыдущие расходы ← сам общие траты на: причина.
  расход от: причина пом: предыдущие расходы + количество.
  количество наличных ← количество наличных − количество.
```

Во время выполнения этого метода первое что происходит это посылка сообщения *общие траты на:* к тому же объекту (себе) который получил *потратить:на:*. Результату этого сообщения посылается сообщение *+ количество*, и результат этого сообщения используется как второй аргумент сообщения *от:пом:*.

Псевдо переменная *сам* может быть использована для реализации рекурсивных функций. Например, сообщение *факториал* понимается целыми числами чтобы вычислить соответствующую функцию. Метод связанным с селектором факториал:

```
факториал
  сам = 0 истина: [ ↑ 1. ].
  сам < 0
    истина: [ сам ошибка: 'factorial invalid'. ]
     ложь: [ ↑ сам * (сам − 1) факториал. ].
```

Получатель это *Целое*. Первое предложение проверяет не является ли получатель 0 и если это так то возвращает 1. Второе предложение проверяет знак получателя, т.к. если он меньше нуля нужно вызвать сообщение об ошибке (все объекты отвечают на сообщение *ошибка:* уведомлением о том что произошла ошибка). Если получатель больше нуля, то возвращённым значением будет:

```
сам * (сам — 1) факториал
```

Значение возвращённое получателем это получатель умноженный на факториал числа на единицу меньшего чем получатель.

3.4.4 Временные переменные

Имена аргументов и сам доступны только в течении выполнения метода. В дополнение к этим переменным метод может иметь несколько других переменных для использования в течении выполнения. Они называются временными переменными. Временные переменные объявляются между образцом метода и предложениями метода. Объявление временных состоит из набора имён переменных между вертикальными чертами.

Значение временной переменной доступно только для предложений метода и уничтожается когда выполнение метода завершается. Все временные переменные изначально ссылаются на пусто. Аргументы сообщения тоже являются временными переменными с уже присвоенными значениями.

В системе Смолток программист может протестировать алгоритм используя временные переменные. Тест можно выполнить используя вертикальные черты для объявления переменных только на время выполнения. Предположим что предложение которое нужно проверить включает ссылку на переменную список. Если переменная список не определена то попытка выполнения предложения приведёт к синтаксической ошибке. Поэтому программист может определить переменную список как временную переменную с помощью добавления перед предложением выражения | список |. Предложения отделяются с помощью точек, так же как и в методах.

```
| список |
список ← Ряд новый: 2.
```

```
список от: 1 пом: 'один'.
список от: 2 пом: 'четыре'.
список цепь для печати.
```

Программист выделяет все пять строк — определение и предложения — и запрашивает их выполнение. Переменная список доступна только во время единичного выполнения выделенного текста.

3.4.5 Элементарные методы

Когда объект получает сообщение от обычно просто посылает другие сообщения, и где же происходят реальные действия? Объект может при получении сообщения изменить значения своих переменных экземпляра, что естественно оценивается как "что-то произошло". Но этого не достаточно. Всё поведение системы вызывается сообщениями, однако не все сообщения отвечают при помощи выполнения методов Смолтока. Есть около сотни примитивных методов способ выполнения которых знает виртуальная машина Смолтока. Примерами сообщения которые выполняются примитивом является сообщение + к малому целому, сообщение *от:* к объектам с нумерованными переменными, и сообщение *новый:* к классам. Когда 3 получает сообщение + *4*, она не выполняет метод Смолтока. Примитивный метод возвращает 7 как значение сообщения. Полный набор примитивных методов включён в четвёртую часть данной книги, которая описывает виртуальную машину.

Метод который реализован как примитивный начинается с выражения вида

<примитив №>

где № это номер указывающий какой примитивный метод будет использоваться. Если примитив не может завершиться правильно, управление передаётся методу Смолтока. За выражением <примитив №> следуют предложения Смолтока которые обрабатывают ошибочную ситуацию.

3.5 Сводка терминов

класс — объект который описывает реализацию набора подобных объектов.
экземпляр — один из объектов описываемых классом, он имеет память и отвечает на сообщения.
переменная экземпляра — переменная доступная единственному объекту во время всей его жизни, переменные экземпляра могут быть именованными или нумерованными.
описание протокола — описание класса в терминах протокола сообщений экземпляра.
описание реализации — описание класса в терминах собственных переменных экземпляра и набора методов которые описывают способ выполнения операций.
образец сообщения — селектор сообщения и набор имён аргументов, по одному имени для каждого аргумента которые должно иметь сообщение с данным селектором.
временная переменная — переменная созданная для выполнения специальной задачи и доступная только в течении этого действия.
переменная класса — переменная разделяемая экземплярами одного класса.
глобальная переменная — переменная разделяемая всеми экземплярами всех классов.
переменная пула — переменная разделяемая экземплярами набора классов.
Смолток — пул разделяемый всеми классами который содержит все глобальные переменные.
метод — описание процедуры выполнения одной из операций объекта; состоит из образца сообщения, объявления временных переменных и последовательности предложений. Метод вызывается когда сообщение совпадающее с образцом посылается экземпляру класса в котором находится метод.
имя аргумента — имя псевдо переменной доступной только в течении выполнения метода; значение имени аргумента это аргумент сообщения которое выполняет метод.
↑ — когда используется в методе, указывает на то что значение следующего предложения это значение возвращаемое методом.

сам — псевдо переменная ссылающаяся на получателя сообщения.
категория сообщения — группа методов в описании класса.
примитивный метод — операция выполняемая напрямую виртуальной машиной Смолтока, не описывается как последовательность предложений на Смолтоке.

Глава 4

Подклассы

Оглавление

Каждый объект в системе Смолток это экземпляр класса. Все экземпляры класса представляют один и тот же вид компонентов. Например, каждый экземпляр *Прямоугольника* представляет прямоугольную область и каждый экземпляр *Словаря* представляет набор связей между именами и значениями. Тот факт что все экземпляры класса представляют один и тот же вид компонента отражается в способе которым экземпляры отвечают на сообщения и в форме их переменных экземпляра.

- все экземпляры класса отвечают на один и тот же набор сообщений и используют один и тот же набор методов для этого.
- все экземпляры класса имеют одно и то же количество именованных переменных экземпляра и используют одни и те же

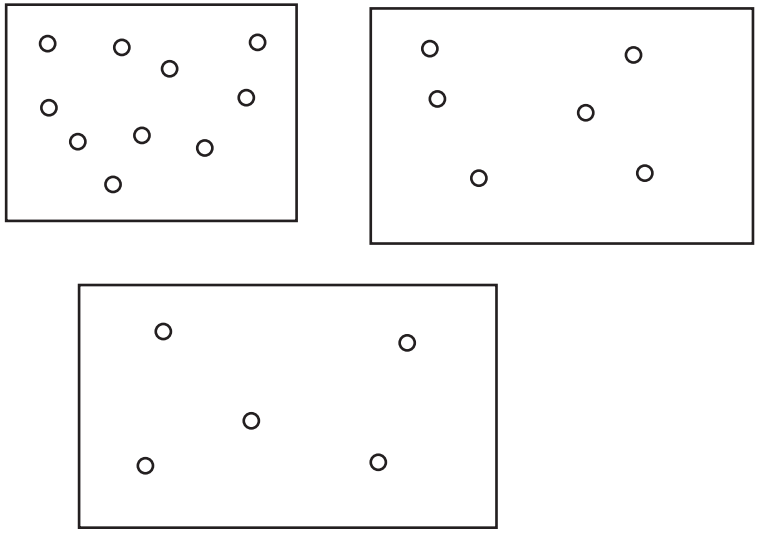


Рис. 4.1

имена для ссылки на них.

- объект может иметь нумерованные переменные только если их могут иметь все экземпляры класса.

Структура классов описанная до сих пор не предоставляет возможности объектам принадлежать к нескольким классам. Каждый объект это экземпляр только одного класса. Эта структура показаны на рисунке 4.1. На рисунке маленькими кружками представлены экземпляры и прямоугольниками классы. Если кружок находится в прямоугольнике то он представляет экземпляр класса представленного прямоугольником.

Отсутствие пересечений между членами классов это ограничение в структуре объектно-ориентированной системы т.к. не позволяется пересечений между описаниями классов. Мы можем захотеть иметь два почти идентичных объекта, но различающихся отдельными чертами. Например числа с плавающей точкой и целые числа похожи возможностью отвечать на арифметические сообщения, но

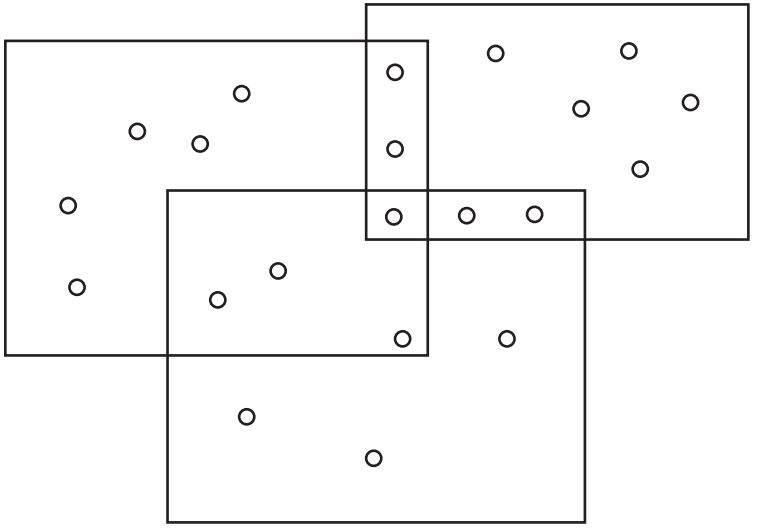


Рис. 4.2

они отличаются способом представления численных значений. *Упорядоченная совокупность* и *Мешок* похожи тем что они являются хранилищами для объектов которые можно добавлять и удалять, но они различаются способом доступа к индивидуальному элементу. Различия между другими похожими объектами могут быть видны снаружи, такие как возможность отвечать на некоторые отличные сообщения или различия могут быть полностью внутренними, такими как ответ на одно и то же сообщения с помощью выполнения различных методов. Если не допускается принадлежность к различным классам то система не может гарантировать данные типы похожести между двумя объектами.

Наиболее общий способ преодолеть данное ограничение это позволить произвольное пересечение между классами (рисунок 4.2).

Мы называем этот подход множественным наследованием. Множественное наследование позволяет ситуации когда объект является экземпляром нескольких классов, в то время как другие объекты это экземпляры только одного или другого класса. Менее общее огра-

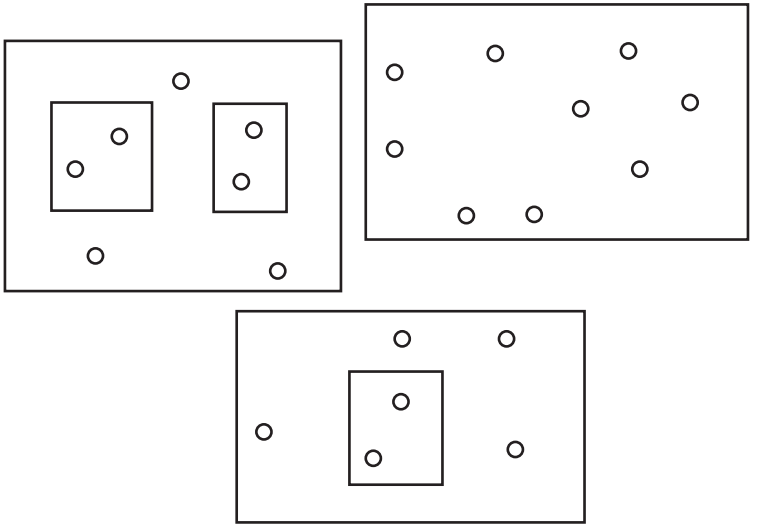


Рис. 4.3

ничение чем не пересечение границ класса это позволить классам включать все экземпляры другого класса, но не позволять более общего разделения (рисунок 4.3).

Мы называем данный подход созданием подклассов. Данное название следует терминологии языка Симула, в котором есть подобная концепция. Подклассы строго ирархичны, если любой экземпляр класса также экземпляр другого класса, то все экземпляры данного класса должны также быть экземплярами другого класса.

Система Смолток предоставляет возможность создания подклассов в форме наследования от классов. Данная глава описывает как подклассы изменяют свои надклассы, как это влияет на связь сообщений с методами и как механизм подклассов предоставляет каркас для классов системы.

4.1 Описание подкласса

Подкласс определяет что его экземпляры должны быть такими же как и экземпляры другого класса, называемого надклассом, за исключением различий которые явно указаны. Программист Смолтока всегда создаёт новый класс как подкласс существующего класса. Класс системы называемый *Объект* описывает общие черты всех объектов системы, поэтому каждый класс должен быть по крайней мере подклассом *Объекта*. Описание (протокола или реализации) класса определяет чем его экземпляры отличаются от экземпляров надкласса. На экземпляры надкласса не может влиять существование подклассов.

Подкласс это тоже класс и поэтому тоже может иметь свои подклассы. Каждый класс имеет один надкласс, однако многие классы могут разделять один и тот же надкласс, т.е. классы образуют древовидную структуру. Класс имеет цепь классов от которых он наследует переменные и методы. Эта цепь начинается с его надкласса и продолжается надклассом надкласса, и т.д.. Цепь наследования продолжается до тех пор пока не дойдёт до класса *Объект*. *Объект* это единственный корневой класс, это единственный класс без надкласса.

Напомним что описание реализации содержит три основных части:

1. Имя класса
2. Объявление переменных
3. Набор методов

Подкласс должен иметь своё имя, но он наследует и объявления переменных и методы своего надкласса. Подкласс может добавить новые переменные и новые методы. Если в подклассе добавлено объявление имён переменных экземпляра, то экземпляры подкласса будут иметь больше переменных экземпляра чем экземпляры надкласса. Если добавлены разделяемые переменные, то они будут доступны экземплярам подкласса, но не экземплярам надкласса. Все имена переменных должны отличаться от имён объявленных в надклассе.

Если класс не имеет нумерованных переменных экземпляра, подкласс может объявить что у его экземпляров будут нумерованные переменные, эти переменные будут добавлены ко всем унаследованным переменным экземпляра. Если класс имеет нумерованные переменные экземпляра его подкласс тоже должен иметь нумерованные переменные экземпляра, подкласс также может добавить новые именованные переменные экземпляра.

Если подкласс добавляет метод с образцом имеющим селектор совпадающий с селектором метода в надклассе, то его экземпляры будут отвечать на сообщение с данным селектором выполняя новый метод. Это называется переопределение метода. Если подкласс добавляет метод с селектором не содержащимся в надклассе, то экземпляр подкласса будет отвечать на сообщения которые не понимают экземпляры надкласса.

Подвидём итоги, каждая часть описания реализации может быть изменена подклассом различными способами:

1. Имя класса должно быть переопределено
2. Можно добавлять переменные
3. Методы могут быть добавлены или переопределены

4.2 Пример подкласса

Описание реализации содержит раздел не указанный в предыдущих главах, он определяет надкласс. В следующем примере описан класс созданный как подкласс класса *Финансовая история* введённого в третьей главе. Экземпляры подкласса разделяют функции *Финансовой истории* для хранения информации о поступлении и трате денег. Они имеет дополнительные функции для отслеживания трат которые облагаются налогом. Подкласс определяет обязательное новое имя для класса (*Налоговая история*), и добавляет одну переменную экземпляра и четыре метода. Один из этих методов (начальный баланс:) переопределяет метод надкласса.

имя класса Налоговая история
надркласс Финансовая история

```
имена переменных экземпляра расходы на налоги

методы экземпляра

запись транзакций

потратить на налоги: количество на: причина
    сам потратить: количество на: причина.
    расходы на налоги ← расходы на налоги + количество.
потратить: количество на: причина вычесть: размер налога
    сам потратить: количество на: причина.
    расходы на налоги ← расходы на налоги + размер налога.

справки

всего налогов
    ↑ расходы на налоги.

инициализация

инициализировать баланс: количество
    над инициализировать баланс: количество.
    расходы на налоги ← 0.
```

Для того чтобы знать все сообщения понимаемые экземплярами *Налоговой истории* нужно просмотреть протоколы классов *Налоговая история*, *Финансовая история* и *Объект*. Экземпляры *Налоговой истории* имеют четыре переменных — три унаследованных от надкласса *Финансовая история*, и одну определённую в классе *Налоговая история*. Класс *Объект* не определяет переменных экземпляра.

Рисунок 4.4 показывает что *Налоговая история* это подкласс *Финансовой истории*. Каждый прямоугольник на данной диаграмме помечен в верхнем левом углу с помщью имени класса который он представляет.

Экземпляры *Налоговой истории* могут быть использованы для записи истории трат таких сущностей траты которых облагаются налогом (люди, домохозяйства, организации). Экземпляры *Финансовой истории* могут быть использованы для записи истории трат

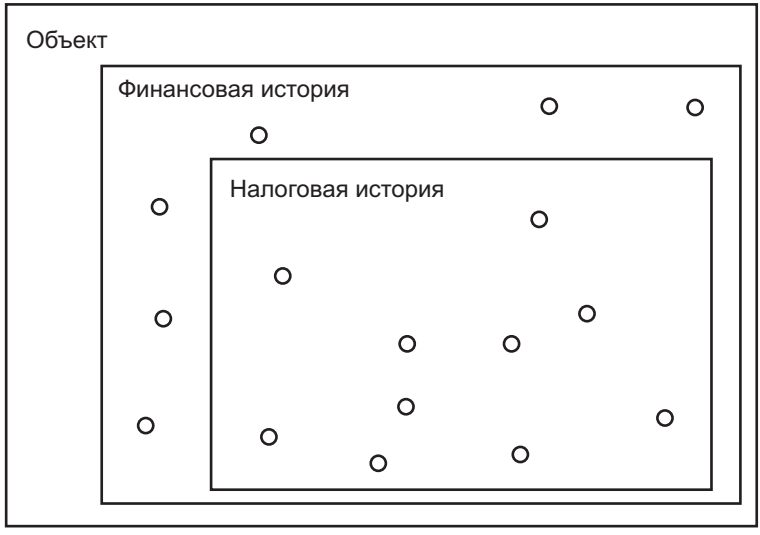


Рис. 4.4

сущностей траты которых не облагаются налогом (благотворительные оргаризации, религиозные организации). В действительности экземпляр *Налоговой истории* может быть использован вместо *Финансовой истории* без видимых эффектов до тех пор пока он отвечает на те же сообщения таким же образом. В добавок к сообщениям наследуемым от *Финансовой истории*, экземпляр *Налоговой истории* может отвечать на сообщения указывающие что все или часть трат облагаются налогом. Новые доступные сообщения: потратить на налоги:на: которое используется если вся сумма это налог и потратить:на:вычесть: которое используется если только часть трат является налогом. Общую сумму налогов можно узнать послав сообщение всего налогов *Налоговой истории*.

4.3 Нахождение метода

При посылке сообщения ищется метод в классе получателя с совпадающим селектором. Если такой не находится то ищется метод в

надклассе класса. Поиск продолжается до тех пор пока не будет найден соответствующий метод. Допустим мы послали экземпляру *Налоговой истории* сообщение с селектором количество наличных. Поиск подходящего метода для выполнения начинается в классе получателя, *Налоговой истории.* Когда он там не находится, то поиск продолжается в надклассе *Налоговой истории*, в *Финансовой истории.* Когда метод с селектором количество наличных находится, то он выполняется в ответ на сообщение. Ответ на данное сообщение возвращает значение переменной экземпляра количество наличных. Это значение находится в получателе сообщения, в нашем случае в экземпляре *Налоговой истории.*

Поиск подходящего метода следует по цепи наследования, обрываясь на классе *Объект.* Если ни одного метода не находится во всех классах цепи наследования, то получателю посылается сообщение не понимаю:; аргумент это не понятое сообщение. Есть метод для селектора не понимаю: в *Объекте* который сообщает программисту об ошибке.

Предположим мы послали экземпляру *Налоговой истории* сообщение с селектором потратить:на:. Этот метод находится в надклассе *Финансовая история.* Метод как он представлен в третьей главе:

```
потратить: количество на: причина
  | предыдущие расходы |
  предыдущие расходы ← сам общие траты на: причина.
  расход от: причина пом: предыдущие расходы + количество.
  количество наличных ← количество наличных − количество.
```

Значения премееных экземпляра (расход и количество наличных) находятся в получателе сообщения, экземпляре *Налоговой истории.* Псевдо переменная сам тоже используется в этом методе; сам пердставляет экземпляр *Налоговой истории* который получил сообщение.

4.3.1 Сообщения себе

Когда метод содержит сообщения чьи получатели это сам, то поиск метода для таких сообщений начинается в классе экземпляра не

смотря на то в каком классе содержится сам метод. Поэтому, когда предложение сам общие траты на: причина выполняется в методе потратить:на: находящемся в Финансовой истории, то поиск метода связанного с селектором сообщения общие траты на: начинается в классе себя, т.е. в Налоговой истории.

Сообщения к себе будут объясняться на примере двух классов с именами *Один* и *Два*. Два это подкласс *Одного* и *Один* это подкласс *Объекта*. Оба класса содержат метод для сообщения тест. Класс *Один* также содержит метод для сообщения результат который возвращает результат предложения сам тест.

```
имя класса Один
надркласс Объект
```

методы экземпляра

пример

тест
  ↑1.
результат 1
  ↑сам тест.

```
имя класса Два
надркласс Один
```

методы экземпляра

пример

тест
  ↑2.

Будет использоваться экземпляр каждого класса для демонстрации нахождения метода для сообщений к себе, пример 1 это экземпляр класса *Один* а пример 2 это экземпляр класса *Два*.

```
пример 1 ← Один новый.
пример 2 ← Два новый.
```

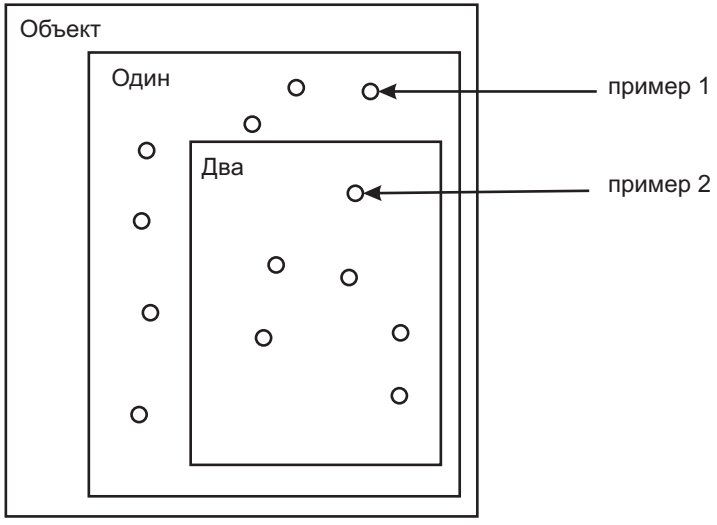


Рис. 4.5

Соотношение между *Один* и *Два* показано на рисунке 4.5. В добавок к именованию прямоугольников для обозначения классов, некоторые кружки также поименованы чтобы указать имена соответствующих экземпляров.

Следующая таблица показывает результаты выполнения различных предложений.

предложение	**результат**
пример 1 тест	1
пример 1 результат 1	1
пример 2 тест	2
пример 2 результат 1	2

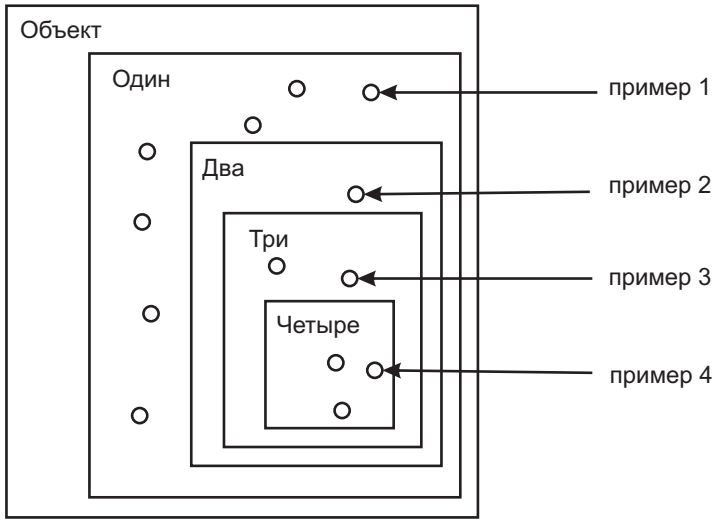


Рис. 4.6

Оба сообщения результат 1 вызывают один и тот же метод, который находится в классе *Один*. Он возвращает различные результаты из за того что сообщения к себе содержится в данном методе. Когда результат 1 посылается примеру 2, то поиск начинается в классе *Два*. Метод не находится в классе *Два*, поэтому поиск продолжается к надклассе, *Один*. Метод для результата находится в классе *Один*, которой содержит одно предложение ↑ сам тест. Псевдо переменная сам ссылается на получателя, пример 2. Поэтому поиск для ответа на тест начинается с класса *Два*. Метод для теста находится в *Два*, который возвращает 2.

4.3.2 Сообщения наду

Дополнительная псевдо переменная именуемая *над* доступна для использования в предложениях метода. Псевдо переменная над ссылается на получателя сообщения, так же как это делает переменная сам. Однако, когда сообщение посылается наду, то поиск метода не

начинается с класса получателя. Вместо этого поиск начинается в надклассе класса содержащего метод. Использование нада позволяет методу получить доступ к методу объявленному в надклассе даже если метод переопределён в подклассе. Использование нада в любом месте отличном от получателя (например как аргумента), ничем не отличается от использования переменной сам; использование нада влияет только на начальный класс с которого начинается поиск метода.

Сообщения наду будут рассмотрены при помощи ещё двух классов именуемых *Три* и *Четыре*. *Четыре* это подкласс *Трёх*, *Три* это подкласс *Двух*, класса из предыдущего примера. Класс *Четыре* переопределяет метод для сообщения тест. Класс *Три* содержит метод для двух новых сообщений - результат 2 возвращает результат предложения сам результат 1, и результат 3 возвращает результат предложения над тест.

имя класса Три
надркласс Два

методы экземпляра

пример

результат 2
↑сам результат 1.
результат 3
↑над тест.

имя класса Четыре
надркласс Три

методы экземпляра

пример

тест
↑4.

Экземпляры классов *Один*, *Два*, *Три* и *Четыре* могут отвечать на сообщения *тест* и *результат*. Ответ экземпляров *Три* и *Четыре* на сообщения показывает эффект производимый надом (4.7).

пример 3 ← Три новый.
пример 4 ← Четыре новый.

Попытка послать сообщения результат 2 или результат 3 примеру 1 или примеру 2 вызовет ошибку т.к. экземпляры классов *Один* и *Два* не понимают сообщений результат 2 и результат 3.

Следующая таблица показывает результаты посылки различных сообщений.

предложение	**результат**
пример 3 тест.	2
пример 4 результат 1.	4
пример 3 результат 2.	2
пример 4 результат 2.	4
пример 3 результат 3.	2
пример 4 результат 3.	2

Когда примеру 3 посылается тест, то используется метод класса *Два*, т.к. в классе *Три* этот метод не переопределён. пример 4 отвечает на результат 4-кой по той же причине почему пример 2 отвечает 2-кой. Когда результат 2 посылается примеру 3, то поиск подходящего метода начинается в классе *Три*. Найденый здесь метод возвращает результат предложения сам результат. Поиск метода для ответа на сообщение результат также начинается в классе *Три*. Подходящий метод не находится в классе *Три* или его надклассе, *Два*. Метод для результата находится в классе *Один* и возвращает результат выполнения предложения сам тест. Поиск для ответа на сообщение тест опять начинается с класса *Три*. В этот раз соответствующий метод находится в надклассе *Три* классе *Два*.

Эффект посылки сообщения наду будут проилюстрирован при помощи ответов на сообщение результат 3 посланных примеру 3 и примеру 4. Когда результат 3 посылается примеру 3, то поиск соответствующего метода начинается в классе *Три*. Метод найденный там возвращает результат предложения над тест. Т.к. тест посыла-

ется наду, то поиск соответствующего метода начинается не в классе *Три*, а в его надклассе, *Два*. Метод тест в классе *Два* возвращает 2. Когда результат 3 посылается примеру 4, то результат опять 2, не смотря на то что класс *Четыре* переопределяет сообщение тест.

Данный пример подчёркивает возможную путаницу: над не начинает поиск в надклассе получателя, который в данном примере равен *Трём*. Сообщение наду начинает поиск в надклассе класса содержащего метод в котором используется над, которым в данном примере является класс *Два*. Даже если *Три* переопределяет метод для теста возвращая 3, результат предложения пример 4 результат 3 будет по прежнему 2. Конечно иногда надкласс класса в котором содержится метод с надом находится в том же надклассе что и надкласс получателя.

Другой пример использования нада даёт метод *Налоговой истории* инициализировать баланс:.

```
инициализировать баланс: количество
    над инициализировать баланс: количество.
    расходы на налоги ← 0.
```

Этот метод переопределяет метод надкласса *Финансовая история*. Метод *Налоговой истории* содержит два предложения. Первое передаёт управление надклассу для инициализации баланса.

```
над инициализировать баланс: количество.
```

Псевдо переменная над ссылается на получателя сообщения, но указывает что поиск метода должен пропустить *Налоговую историю* и начинать с *Финансовой истории*. В данном случае предложения из *Финансовой истории* не будут дублироваться в *Налоговой истории*. Второе предложение данного метода производит инициализацию особую для класса.

```
расходы на налоги ← 0.
```

Если заменить нада на сам в методе инициализировать баланс:, то результатом будет бесконечная рекурсия, т.к. каждый раз при посылке инициализировать баланс: это сообщение будет посылаться ещё раз.

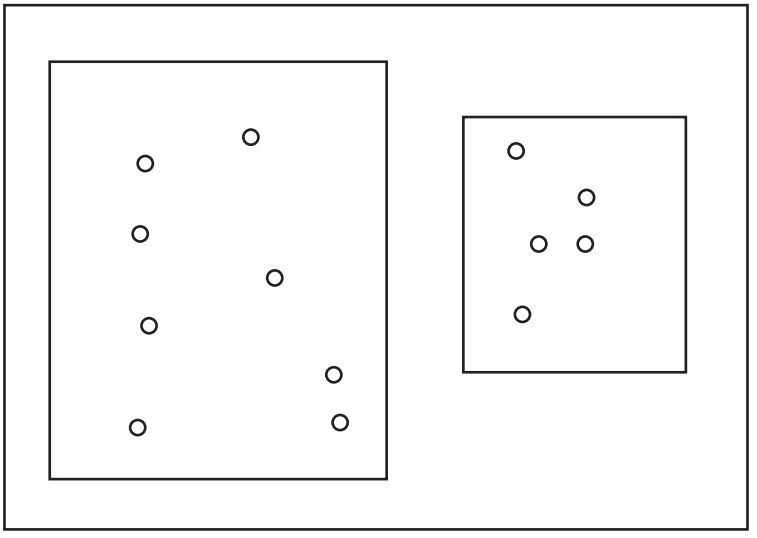


Рис. 4.7

4.4 Абстрактные надклассы

Абстрактные надклассы создаются когда два класса разделяют часть сових описаний и ни тот ни другой не являются подклассом другого. Взаимный надкласс создаётся для двух классов и он содержит их общие черты. Такой вид надклассов называется абстрактным потому что эти классы создаются не для создания экземпляров. В терминах прямоугольников абстрактные надклассы представляют ситуацию показанную на 4.7. Заметьте что абстрактный класс не содержит прямых экземпляров.

В качестве примера использования абстрактного надкласса рассмотри два класса чьи экземпляры представляют словари. Один класс, именуемый *Малый словарь*, минимизирует место требуемое для хранения своего содержания; другой, именуемый *Быстрый словарь*, хранит имена и значения редко и использует технику хэширования для нахождения элементов. Оба класса используют два параллельных списка которые содержат имена и связанные с ними

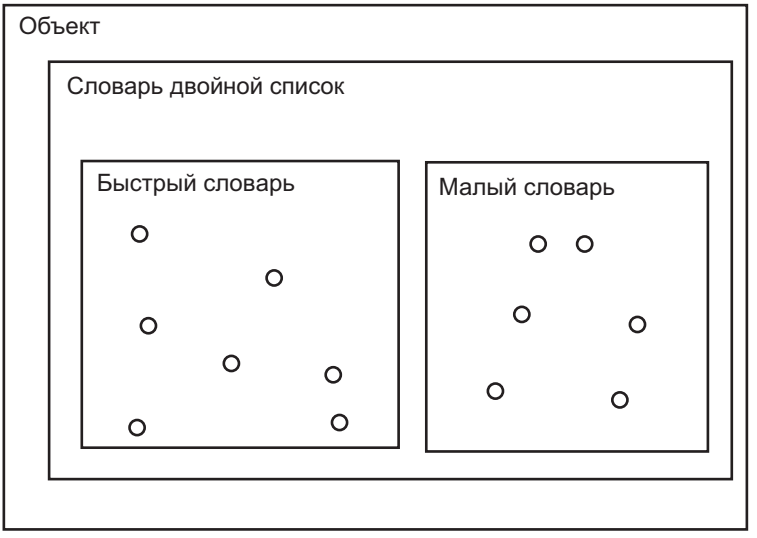


Рис. 4.8

значения. *Малый словарь* помещает имена и значения подряд и использует простой линейный поиск для нахождения имён. *Быстрый словарь* помещает имена и значения раздельно и использует технику хэширования для нахождения имён. За исключением различий запоминания имён эти два класса очень похожи они реализуют один и тот же протокол и они оба используют параллельные списки для запоминания содержания. Эти подобия представлены в абстрактном надклассе именуемом *Словарь двойной список*. Взаимоотношения между этими тремя классами показаны на рисунке 4.8.

Далее показано описание реализации для абстрактного класса *Словарь двойной список*.

имя класса Словарь двойной список

надкласс Объект

имена переменных экземпляра имена значения

методы экземпляра

доступ

```
от: имя
  | номер |
  номер ← сам номер для: имя.
  номер == 0
    истина: [сам ошибка: 'Name not found'.]
    ложь: [↑значения от: номер.].
от: имя пом: значение
  | номер |
  номер ← сам номер для: имя.
  номер == 0 истина: [номер ← сам новый номер для: имя.].
  ↑значения от: номер пом: значение.
```

проверки

```
содержит: имя
  ↑(сам номер для: имя) ~= 0.
пустой
  ↑сам размер == 0.
```

инициализация

```
инициализировать
  имена ← Ряд новый: 0.
  значения ← Ряд новый: 0.
```

Данное описание *Словаря двойного списка* использует только сообщения определённые в самом *Словаре двойном списке* или те которые уже описаны в данной или предыдущих главах. Внешний протокол для *Словаря двойного списка* содержит сообщения *от:*, *от:пом:*, *содержит:*, пустой и инициализировать. Новый экземпляр *Словаря двойного списка* (в действительности экземпляр одного из его подклассов) создаётся посылкой сообщения новый. Затем это сообщение посылает сообщение *инициализировать* чтобы присвоить значения двум переменным экземпляра. Эти переменные изначально являются пустыми рядами (Ряд новый: 0).

Три сообщения *себе* используемые в этих методах не реализованы в *Словаре двойном списке — размер*, *номер для:* и *новый номер для:*. Поэтому класс *Словарь двойной список* называется абстрактным. Если создать экземпляр, то он не сможет правильно отвечать на все сообщения. Два подкласса, *Малый словарь* и *Быстрый словарь* должны реализовать эти три недостающих сообщения. Тот факт что поиск всегда начинается в классе экземпляра на который ссылается *сам* означает что метод в надклассе может объявлять какое сообщение послать *самому*, но соответствующий метод находится в подклассе. Таким образом надкласс может предоставлять каркас для методов которые используются или реализуются в подклассе.

Малый словарь это подкласс *Словаря двойного списка* который использует минимальный объём памяти для хранения связей, но требует большого времени для нахождения ассоциации. Он предоставляет методы для трёх сообщений которые не были реализованы в *Словаре двойном списке — размер*, *номер для:* и *новый номер для:*. Он не добавляет переменных.

имя класса Малый словарь
надркласс Словарь двойной список
нумерованные переменные экземпляра

методы экземпляра

доступ

```
размер
  ↑имена размер.
```

личные

```
номер для: имя
  1
    до: имена размер
  делать: [:номер | (имена от: номер) == имя истина: [↑номер.].].
  ↑0.
новый номер для: имя
```

```
    сам расти.
    имена от: имя размер пом: имя.
    ↑имена размер.
расти
    | старые имена старые значения |
    старые имена ← имена.
    старые значения ← значения.
    имена ← Ряд новый: имена размер + 1.
    значения ← Ряд новый: значения размер + 1.
    имена заменить от: 1 до: старые имена размер на: старые имена.
    значения заменить от: 1 до: старые значения размер на: старые значения.
```

Т.к. имена запоминаются подряд то размер *Малого словаря* это размер его ряда *имена*. Номер конкретного имени определяется при помощи линейного поиска в ряде имён. Если не находится совпадение, то номер это 0, указывает что поиск неудачен. Когда к словарю добавляется новая связь используется метод *новый номер для:* чтобы найти подходящий номер. Этот класс предполагает что размер рядов имена и значения в точности равен количеству текущих хранимых элементов. Сообщение *расти* создаёт два *Ряда* которые копируют предыдущие элементы, которые увеличены на один элемент в конце ряда. В методе для *новый номер для:* сначала увеличиваются размеры имён и значений и затем новое имя добавляется в новую пустую позицию (в последнюю). Метод который вызывает *новый номер для:* отвечает за присвоение значения.

Можно выполнить следующие примеры предложений:

предложение	**результат**
возрасты ← Малый словарь новый.	новый неинициализированный экземпляр
возрасты инициализировать.	переменные экземпляра инициализированы
возрасты пустой.	истина
возрасты от: 'Brett' пом: 3.	3
возрасты от: 'Dave' пом: 30.	30
возрасты содержит: 'Sam'.	ложь

возрасты содержит: 'Brett'.	истина
возрасты размер.	2
возрасты от: 'Dave'.	30

Для каждого вышеприведённого предложения показано в каком классе находится метод и в каком классе ищутся сообщения посылаемые *себе*.

селектор сообщения	**сообщения себе**	**класс метода**
инициализировать		Словарь двойной список
от:пом:		Словарь двойной список
номер для:		Малый словарь
	новый номер для:	Малый словарь
включает:		Словарь двойной список
	номер для:	Малый словарь
размер		Малый словарь
от:		Словарь двойной список
	номер для:	Малый словарь
	ошибка:	Объект

Быстрый словарь это другой подкласс *Словаря двойного списка*. Он использует технику хэширования для размещения имён. Хэширование требует больше памяти, но на поиск тратиться меньше времени чем для линейного поиска. Все объекты отвечают на сообщение *хэш* возвращая число. *Числа* отвечают на сообщение \\ возвращая значение по модулю аргумента.

имя класса Быстрый словарь

надркласс Словарь двойной список

методы экземпляра

доступ

```
размер
  | размер |
  размер ← 0.
  имена делать: [:имя | имя не пусто истина: [размер ← размер + 1.].].
  ↑ размер.
```

инициализация

```
инициализировать
  имена ← Ряд новый: 4.
  значения ← Ряд новый: 4.
```

личные

```
номер для: имя
  | номер |
  номер ← имя хэш \\ имена размер + 1.
  [(имена от: номер) == имя.]
    пока ложь: [
      (имена от: номер) это пусто истина: [↑ 0.].
      номер ← номер \\ имена размер + 1.].
  ↑ номер.
новый номер для: имя
  | номер |
   имена размер − сам размер <= (имена размер / 4) истина: [сам
расти.].
  номер ← имя хэш \\ имена размер + 1.
  [(имена от: номер) это пусто.]
    пока ложь: [номер ← номер \\ имена размер + 1.].
  имена от: номер пом: имя.
  ↑ номер.
расти
  | старые имена старые значения |
  старые имена ← имена.
  старые значения ← значения.
```

```
имена ← Ряд новый: имена размер * 2.
значения ← Ряд новый: значения размер * 2.
1
  до: старые имена размер
  делать: [
    :номер |
    ( старые имена от: номер ) это нуль
      ложь: [
        сам
          от: ( старые имена от: номер )
          пом: ( старые значения от: номер ). ]. ].
```

Быстрый словарь переопределяет реализацию метода *инициализировать* класса *Словарь двойной список* чтобы создать *Ряды* которые уже имееют некоторое выделенное пространство (Ряд новый: 4). Размер *Быстрого словаря* это не просто размер одной из его переменных т.к. *Ряды* всегда содержат пустые элементы. Поэтому размер определяется при помощи проверки каждого элемента *Ряда* и подсчитывается количество не нулевых элементов.

Реализация *нового номера для:* следует той же идее что используется в *Малом словаре* за исключением случая когда размер *Ряда* изменяется (удваивается в случае данного метода расти), каждый элемент явно копируется из старого *Ряда* в новый поэтому элементы хэшируются заново. Размер не всегда изменяется как в *Малом словаре*. Размер *Быстрого словаря* изменяется только когда количество свободной памяти для имён становится меньше минимума. Минимум равен 25 процентам элементов.

```
имена размер − сам размер <= ( имена размер / 4 )
```

4.5 Каркасные сообщения для подкласса

Если следовать стилю программирования, метод не должен включать сообщений к себе если сообщения не реализованы классом или наследуемыми от него подклассами. В описании *Словаря двойного списка* есть три таких сообщения — *размер*, *номер для:* и *новый номер для:*. Как мы увидим в последующих главах возможность

отвечать на сообщение *размер* наследуется от *Объекта*, ответ это число нумерованных переменных. Подразумевается что подклассы *Словаря двойного списка* переопределяют этот метод для возвращения количества имён в словаре.

Специальное сообщение, *ответственность подкласса*, определено в *Объекте*. Оно используется в реализации сообщений которые не могут быть правильно реализованы в абстрактном классе. Поэтому реализации методов *размер*, *номер для:* и *новый номер для:*, по соглашению принятому в Смолтоке, должны состоять из предложения

```
сам ответственность подкласса.
```

Ответ на это сообщение это вызов следующего метода определённого в классе *Объект*.

```
ответственность подкласса
  сам
    ошибка: 'My subclass should have overridden '
      , этот контекст отправитель селектор цепь для печати.
```

В данном случае, если метод должен быть реализован в подклассе абстрактного класса, программисту выводится сообщение об ошибке и как её исправить. Более того, используя данное сообщение программист создаёт абстрактный класс в котором все сообщения посылаемые себе реализованы, и в этих реализациях содержится указание какие методы должны быть переопределены в подклассах.

По соглашению, если программист хочет чтобы сообщение унаследованное от абстрактного надкласса не было реализовано, то подходящий путь для этого переопределить унаследованный метод так

```
сам не должен реализовывать.
```

Ответ на это сообщение это вызов следующего метода определённого в классе *Объект*.

```
не должен реализовывать
  сам ошибка: 'This message is not appropriate for this object'.
```

В системе Смолток есть несколько ирархий подклассов которые используют идею создания каркаса сообщений чьи реализации должны быть закончены в подклассах. Эти классы описывают раз-

личные виды совокупностей (смотри главы 9 и 10). Классы совокупностей организованный в ирархию для того чтобы разделить как можно большее количество подобных описаний классов совокупностей. Они используют сообщения *ответственность подкласса* и *не должен реализовывать*. Другой пример использования подклассов это ирархии одномерных величин и чисел (смотри главы 7 и 8).

4.6 Сводка терминов

подкласс — класс который наследует переменные и методы от существующего класса.
надкласс — класс от которого наследуются методи и переменные.
Объект — класс являющийся корнем дерева ирархии наследования классов.
переопределение метода — определение метода в подклассе для сообщения уже определённого в надклассе.
над — псевдо переменная ссылающаяся на получетел сообщения; отличается от сам способом поиска метода.
абстрактный класс — класс определяющий протокол, но не полностью его реализующий, по соглащению его экземпляры не создаются.
ответственность подкласса — сообщение для указания ошибки, подкласс должен определить одно из сообщений надкласса.
не должен реализовывать — сообщения для указания ошибки, сообщение наследуется от надкласса но явно не доступно для экземпляров подкласса.

Глава 5

Метаклассы

Оглавление

Т.к. все компоненты системы Смолток представляются объектами и все объекты это экземпляры класса, сами классы должны быть представлены экземплярами класса. Класс чьи экземпляры сами являются классами называются метаклассами. Эта глава описывает специальные свойства метаклассов. Примеры показывают как используются метаклассы для создания экземпляров и общих запросов к классам.

В ранних версиях системы Смолток был только один метакласс называемый Класс. Он отвечал за организацию класса показанную на рисунке 5.1. Как и в главе 4 прямоугольники обозначают классы а кружки обозначают экземпляры класса в котором они содержатся. Там где возможно прямоугольники обозначаются с помощью имени класса. Заметьте что есть по одному кружку в прямоугольнике Класс для каждого прямоугольника на рисунке.

При данном подходе возникают трудности т.к. протокол сообще-

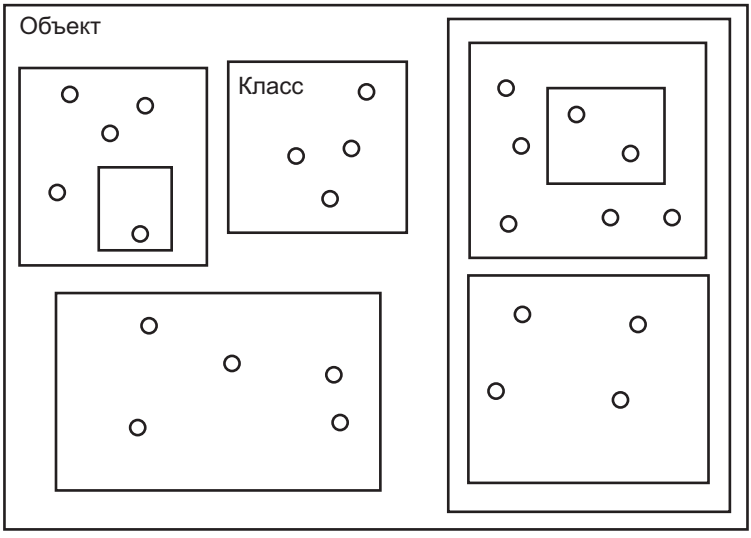


Рис. 5.1

ний всех классов должен быть одинаковым т.к. он определён в одном месте. Практически сообщения используемые для создания новых экземпляров были одними и теми же для всех классов и не могли произвести специальной инициализации. С одним метаклассом, все классы отвечали на сообщение новый или новый: возвращая экземпляр чьи переменные ссылались на *пусто*. Для большинства объектов *пусто* это не подходящее значение переменной экземпляра, поэтому новые экземпляры инициализировались при помощи другого сообщения. Программист должен был удостовериться что каждый раз когда посылалось сообщение новый или новый: другое сообщение посылалось для правильной инициализации. Примеры такова вида инициализации были показаны в главе 4 для *Малого словаря* и *Финансовой истории*.

Система Смолток убрала ограничение на то что все классы используют одни и те же сообщения для создания экземпляров сделав каждый класс экземпляром своего собственного метакласса. Когда создаётся новый класс, для него автоматически создаётся новый ме-

такласс. Метаклассы подобны другим классам т.к. они содержат методы используемые их экземплярами. Метаклассы отличаются от других классов потому что они не являются экземплярами метаклассов. Вместо этого все они являются экземплярами класса называемого *Метакласс*. Также метаклассы не имеют имени класса. Метаклассы доступны при помощи посылки сообщения класс их экземпляру. Например, метакласс *Прямоугольника* доступен при помощи предложения *Прямоугольник класс*.

Сообщения метакласса обычно поддерживают создание и инициализацию экземплров, и инициализации переменных класса.

5.1 Инициализация экземпляров

Каждый класс может отвечать на сообщения которые запрашивают новые инициализированные экземпляры. Нужны различные метаклассы потому что инициализирующие сообщения различны для различных классов. Например, мы уже видели что *Время* создаёт новые экземпляры в ответ на сообщение *текущее* и *Дата* создаёт новые экземпляры в ответ на сообщение *сегодня*.

```
Время текущее.
Дата сегодня.
```

Эти сообщения не понятны для *Точки*, класса чьи экземпляры представляют двумерную точку. *Точка* создаёт новые экземпляры в ответ на сообщения с селектором *икс:игрек:* и двумя аргументами определяющими координаты. В свою очередь это сообщение не понятно для *Времени* и *Даты*.

```
Точка икс: 100 игрек: 150.
```

Класс *Прямоугольник* понимает несколько сообщений которые создают экземпляры. Сообщение с селектором *начало:угол:* принимает *Точки* представляющие верхний левый и нижний правый углы в качестве аргументов.

```
Прямоугольник
  начало: ( Точка икс: 50 игрек: 50 )
  угол: ( Точка икс: 250 игрек: 300 ).
```

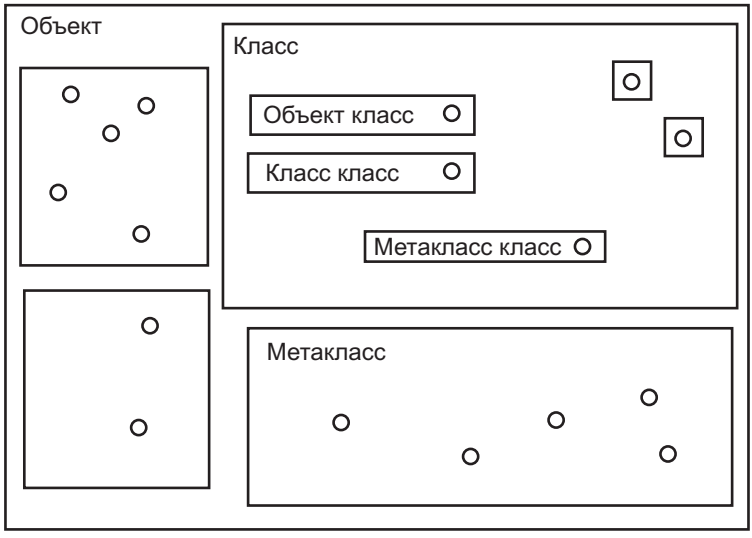


Рис. 5.2

Сообщение с селектором *начало:размеры:* получает в качестве аргументов верхний левый угол и *Точку* представляющую ширину и высоту. Тот же прямоугольник может быть создан следующим предложением.

```
Прямоугольник
    начало: ( Точка икс: 50 игрек: 50 )
    размеры: ( Точка икс: 200 игрек: 250 ).
```

В системе Смолток *Класс* это абстрактный надкласс для всех метаклассов. *Класс* описывает общие свойства классов. Каждый метакласс добавляет поведение особенное для данного экземпляра. Метаклассы могут добавлять новые сообщения создания экземпляров как это делают *Дата*, *Время*, *Точка* и *Прямоугольник*, или они могут переопределять сообщения *новый* и *новый:* для реализации некоторой инициализации по умолчанию.

Организация классов и экземпляров системы, как было описано выше, представлена на рисунке 5.2.

На этом рисунке мы видим классы *Объект*, *Метакласс*, *Класс* и метаклассы для каждого из них. Каждый кружок в прямоугольнике *Метакласс* обозначает метакласс. Каждый кружок в прямоугольнике *Класс* обозначает подкласс *Класса*. Есть один прямоугольник для каждого кружка в прямоугольнике *Метакласс*. Каждый из этих прямоугольников содержит кружки обозначающие его экземпляры; эти экземпляры ссылаются на *Объект* или один из подкласс *Объекта*, но не на метаклассы.

5.2 Пример метакласса

Т.к. есть соответствие один к одному между классом и его метаклассом, то их описания представляются вместе. Описание реализации включает часть называемую «методы класса» которая показывает методы добавленные метаклассом. Протокол для метакласса всегда находится при просмотре методов класса описания реализации его единственного экземпляра. Таким образом, сообщения посланные классу (методы класса) и сообщения посланные экземпляру класса (методы экземпляра) перечисляются вместе как части полного описания реализации.

Следующая новая версия описания реализации *Финансовой истории* включает методы класса.

имя класса Финансовая история
надркласс Объект
имена переменных экземпляра количество наличных приход расход

методы класса

создание экземпляра

инициализировать баланс: **количество**
↑над новый инициализировать баланс: количество.

методы экземпляра

запись транзакций

получить: **количество** **из:** **источник**

```
    приход от: источник пом: (сам общее поступление из: источник) + количество.
  количество наличных ← количество наличных + количество.
потратить: количество на: причина
  | предыдущие расходы |
  предыдущие расходы ← сам общие траты на: причина.
  расход от: причина пом: предыдущие расходы + количество.
  количество наличных ← количество наличных − количество.

справки

количество наличных
  ↑количество наличных.
общее поступление из: источник
  (приход содержит ключ: источник)
    истина: [↑приход от: источник.]
    ложь: [↑0.].
общие траты на: причина
  (расход содержит ключ: причина)
    истина: [↑расход от: причина.]
    ложь: [↑0.].

инициализация

инициализировать баланс: количество
  количество наличных ← количество.
  приход ← Словарь новый.
  расход ← Словарь новый.
```

Данный пример показывает как метакласс создаёт инициализированные экземпляры. Методы создания экземпляров не имеют прямого доступа к переменным экземпляра нового экземпляра (количество наличных, приход и расход). Это происходит из за того что методы создания экземпляра это не часть экземпляра класса, а часть класса класса. Поэтому методы создания экземпляров сначала создают неинициализированный экземпляр и затем посылают инициализирующее сообщение, *инициализирвоатъ баланс:*, новому экземпляру. Метод для этого сообщения находится в части методов

экземпляра *Финансовой истории*; он может присваивать соответствующие значения переменным экземпляра. Обычно это сообщение посылается один раз и только методами класса.

5.3 Наследование метаклассов

Подобно другим классам, метаклассы наследуют от надклассов. Простейший способ организовать наследование метаклассов это сделать каждый из них подклассом *Класса*. Данная организация показана на рисунке 5.2. *Класс* описывает общие свойства классов. Каждый метакласс добавляет поведение особенное для своего экземпляра. Метаклассы могут добавлять новые сообщения создания экземпляров или могут переопределять сообщения *новый* и *новый:* для выполнения некоторой инициализации по умолчанию.

Когда метакласс добавляется к системе Смолток, происходит ещё одно действие в организации классов. Ирархия подклассов метакласса создаётся параллельно ирархии подклассов класса который является экземпляром метакласса. Т.е. если *Налоговая история* это подкласс *Финансовой истории*, то метакласс *Налоговой истории* должен стать подклассом метакласса *Финансовой истории*. Метакласс всегда имеет только один экземпляр.

Абстрактный класс *Описание класса* предоставляет описание классов и их экземпляров. *Класс* и *Метакласс* это подклассы *Описания класса*. Т.к. цепь наследования всех объектов заканчивается *Объектом* и *Объект* не имеет надкласса, то надкласс метакласса *Объекта* это *Класс*. От *Класса* метаклассы наследуют сообщения которые предоставляют протокол для создания экземпляров (рисунок 5.3).

Цепь надклассов от *Класса* доходит в конечном счёте до *Объекта*. Заметьте что ирархия прямоугольников с именем класс Объекта подобна ирархии прямоугольнков с именем Объект; это подобие иллюстрирует параллельность наследования. Полное описание данной части системы, влючающие взаимоотношения между *Метаклассом* и его метаклассами представлено в главе 16.

В качестве примера наследования метаклассов рассмотрим реализацию *инициализировать баланс:* в классе *Финансовая история*.

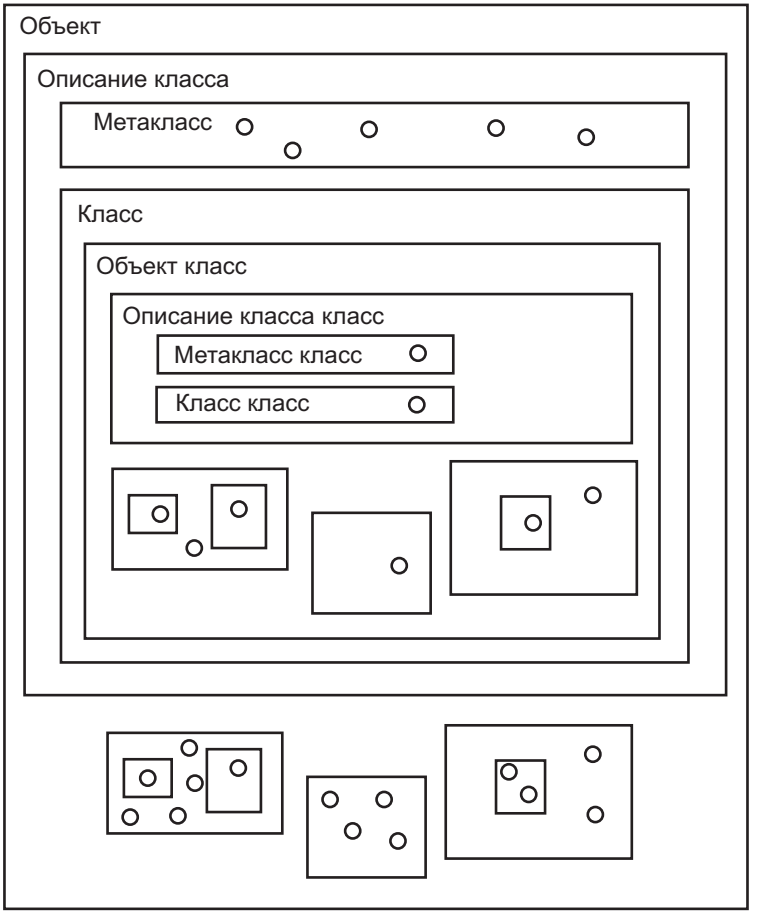

Объект
Описание класса
Метакласс
Класс
Объект класс
Описание класса класс
Метакласс класс
Класс класс

Рис. 5.3

инициализировать баланс: количество
↑над новый инициализировать баланс: количество.

Этот метод создаёт экземпляр выполняя предложение *над новый*; оно использует метод *новый* находящийся в методах класса надкласса, не в методах класса находящихся в данном классе. Заметем он посылает новому экземпляру сообщение *инициализировать баланс:* с начальным количеством денег в виде аргумента.

Если выполнить предложение:

Финансовая история инициализировать баланс: 350.

поиск метода для ответа на *инициализировать баланс:* начнётся в классе *Финансовой истории*, т.е. в методах класса *Финансовой истории*. Метод с таким селектором находится здесь. Метод содержит два сообщения:

1. Посылка *наду* сообщения новый.

2. Посылка результату первого пункта сообщения *инициализировать баланс:*.

Поиск метода новый начинается в надклассе класса *Финансовой истории*, в данном случае, в классе *Объекта*. Метод здесь не находится, поэтому поиск продолжается в следующем надклассе содержащесмся в цепи наследования т.е. в *Классе*. Сообщение находится в *Классе*, и оно выполняется используя примитивный метод. Реузльтат это неинициализированный экземпляр *Финансовой истории*. Затем этому экземпляру посылается сообщение *инициализировать баланс:*. Поиск метода для ответа начинается в классе экземпляра, т.е. в *Финансовой истории* (в методах экземпляра). Метод находится и при выполнении присваивает значение каждой переменной экземпляра.

5.4 Инициализация переменных класса

Главный способ использования сообщений классу после создания экземпляров это инициализация переменных класса. Реализация описаний переменных даёт им только имена, но не значения.

Когда класс создаётся, то создаются именованные переменные класса, но они все имеют значение *пусто*. Обычно метакласс определяет метод который инициализирует переменные класса. По соглашению, метод инициализации переменных класса обычно связан с унарным сообщением *инициализировать* в категории инициализация класса.

Переменные класса доступны и классу и его метаклассу. Присваивание значения переменной класса может быть выполнено в методе метакласса, в отличии от непрямого способа при помощи сообщения для переменных экземпляра.

Ниже показан пример *Налоговой истории*, в этот раз с переменными класса которые нужно инициализировать. *Налоговая история* это подкласс *Финансовой истории*. Он определяет одну переменную класса, *Минимальный налог*.

имя класса Налоговая история
надркласс Финансовая история
имена переменных экземпляра расходы на налоги
имена переменных класса Минимальный налог

методы класса

создание экземпляра

```
инициализировать баланс: количество
  | новая история |
  новая история ← над инициализировать баланс: количество.
  новая история инициализировать налоги.
  ↑ новая история.
новый
  | новая история |
  новая история ← над инициализировать баланс: 0.
  новая история инициализировать налоги.
  ↑ новая история.
```

инициализация класса

инициализировать

```
    Минимальный налог ← 2300.
```

методы экземпляра

запись транзакций

```
потратить на налоги: количество на: причина
    сам потратить: количество на: причина.
    расходы на налоги ← расходы на налоги + количество.
потратить: количество на: причина вычесть: размер налога
    сам потратить: количество на: причина.
    расходы на налоги ← расходы на налоги + размер налога.
```

справки

```
детализированный
    ↑ расходы на налоги >= Минимальный налог.
всего налогов
    ↑ расходы на налоги.
```

собственные

```
инициализировать налоги
    расходы на налоги ← 0.
```

Эта версия *Налоговой истории* добавляет пять методов экземпляра, один из которых это *детализированный*. Ответом на которое является истина или ложь в зависимости от того накопилось ли достаточно налогов чтобы детально их отражать в отчёте. Налоговое законодательство определяет минимальное количество налогов в размере 2300, поэтому если общее количество налогов меньше этой суммы, то следует использовать standard deduction. На константу 2300 ссылается переменная класса *Минимальный налог*. Чтобы посылка собщения *детализированный* экземпляру *Налоговой истории* прошла успешно, переменная класса *Минимальный налог* должна иметь численное значение. Это делается при помощи посылки сообщения *инициализировать* классу перед созданием экземпляров.

```
Налоговая история инициализировать.
```

Это сообщение посылается только один раз, после того как сообщение класса *инициализировать* будет определено. Переменная разделяется всеми экземплярами класса.

В соответствии с приведённым описанием класса новый экземпляр *Налоговой истории* может быть создан при помощи сообщения классу *инициализировать баланс:* или *новый*. Предположим выполняется предложение:

```
Налоговая история инициализировать баланс: 100.
```

Определение какие методы будут использоваться при выполнении предложения зависит от классов/надклассов в цепи наследования *Налоговой истории*. Метод *инициализировать баланс:* находится в методах класса *Налоговой истории*.

```
инициализировать баланс: количество
  | новая история |
  новая история ← над инициализировать баланс: количество.
  новая история инициализировать налоги.
  ↑ новая история.
```

Этот метод объявляет временную переменную *новая история*. Первое предложение метода это присваивание этой временной переменной.

```
новая история ← над инициализировать баланс: количество.
```

Псевдо переменная *над* ссылается на получателя. Получатель это класс *Налоговая история*, его класс это метакласс. надкласс метакласса это метакласс *Финансовой истории*. Здесь мы будем искать метод который будет выполнять сообщение. Метод *инициализировать баланс:*

```
инициализировать баланс: количество
  ↑ над новый инициализировать баланс: количество.
```

Мы уже рассматривали выполнение этого метода. Ответ на сообщение *новый* находится в *Классе*. Новый экземпляр получателя, *Налоговой истории*, создаётся и ему посылается сообщение *инициализировать баланс:*. Поиск метода *инициализировать баланс:*

начинается в классе нового экземпляра, т.е. в *Налоговой истории*. Там он не находится. Поиск продолжается в надклассе *Налоговой истории*. Он находится и выполняется. Переменным экземпляра описанным в *Финансовой истории* присваиваются значения. Затем значение первого предложения метода класса для селектора *инициализировать баланс:* в *Налоговой истории* частично инициализирует новый экземпляр. Этот новый экземпляр присваивается временной переменной *новая история*.

Затем новой истории посылается сообщение инициализировать налоги. Поиск начинается в классе получателя, *новой истории*; класс это *Налоговая история*. Метод находится. Он присваивает значение четвёртой переменной экземпляра значение 0.

Третье предложение сообщения создания экземпляра это возвращение нового экземпляра.

Далее представлен альтернативный способ реализации класса налоговая история. В этом альтернативном описании, методы создания экземпляра не переопределяются. Вместо этого сообщение *инициализировать баланс:* переопределено чтобы иметь доступ к дополнительным переменным экземпляра.

имя класса Налоговая история
надркласс Финансовая история
имена переменных экземпляра расходы на налоги
имена переменных класса Минимальный налог

методы класса

инициализация класса

инициализировать
Минимальный налог ← 2300.

методы экземпляра

запись транзакций

потратить на налоги: количество на: причина
сам потратить: количество на: причина.
расходы на налоги ← расходы на налоги + количество.

```
потратить: количество на: причина вычесть: размер налога
    сам потратить: количество на: причина.
    расходы на налоги ← расходы на налоги + размер налога.

    справки

детализированный
    ↑ расходы на налоги >= Минимальный налог.
всего налогов
    ↑ расходы на налоги.

    собственные

инициализировать баланс: количество
    над инициализировать баланс: количество.
    расходы на налоги ← 0.
```

Используя это альтернативное описание для *Налоговой истории*, выполнение сообщения *инициализировать баланс:* в предложении:

```
Налоговая история инициализировать баланс: 350
```

вызывает поиск метода в классе *Налоговая история*. Там он не находится. Поиск продолжается в надклассе, в классе *Финансовая история*. Здесь он находится. Выполнение метода содержит следующее предложение:

```
над новый инициализировать баланс: количество.
```

Метод для сообщения новый находится в *Классе*. Поиск метода *инициализировать баланс:* начинается в классе нового экземпляра, в *Налоговой истории*. Метод для *инициализировать баланс:* находится в *Налоговой истории*. Ответ на *инициализировать баланс:* в *Налоговой истории* это посылка того же сообщения наду, поэтому поиск метода начинается в *Финансовой истории*. Метод находится и трём переменным экземпляра присваивается значение. Второе предложение метода *инициализировать баланс:* в *Налоговой истории* присваивает значение 0 четвёртой переменной. Ре-

зультат изначального сообщения это полностью инициализированный экземпляр *Налоговой истории*.

5.5 Краткое изложение поиска метода

Определение действий которые происходят при посылки сообщения включает в себя поиск метода в ирархии классов получателя. Поиск начинается с класса получателя и следует по цепи наследования. Если метод не находится после поиска в последнем надклассе, *Объекте*, то выдаётся сообщение об ошибке. Если получатель это класс, то его класс это надкласс. Сообщения на которые класс может отвечать приводятся в описании реализации в разделе «методы класса». Если получатель это не класс, то сообщения на которые он может отвечать приводятся в описании реализации в части «методы экземпляра».

Псевдо переменная *сам* ссылается на получателя сообщения для которого выполняется метод. Поиск метода для сообщения *себе* начинается в классе *себя*. Псевдо переменная *над* также ссылается на получателя сообщения. Поиск метода для сообщения посланного *наду* начинается в надклассе класса в котором находится выполняемый метод.

На этом описание языка Смолток заканчивается. Для использования системы программист должен иметь знания об основных классах системы. Вторая часть даёт детальное описание протокола сообщений для каждого класса системы и приводит примеры, часто показывая описание реализации классов системы. Третья часть вводит пример программы обозримого размера. Перед тем как перейти к классам системы, читатель может перепрыгнуть к третьей части чтобы почувствовать как пишутся большие программы.

5.6 Сводка терминов

метакласс — класс класса.
Класс — абстрактный надкласс для всех классов отличных от метаклассов.
Метакласс — класс чьи экземпляры это классы классов.

Часть II

Обзор функций системы

В первой части был представлен обзор языка Смолток с двух точек зрения: с точки зрения семантики объектов и посылки сообщений и с точки зрения синтаксиса предложений языка. Программист использующий Смолток сначала должен понять семантику языка: вся информация представляется в форме объектов и что все действия совершаются с помощью посылки сообщений объектам. Каждый объект описывается классом; каждый класс, за исключением класса *Объект*, это подкласс другого класса. Программирование системы Смолток включает описание классов новых объектов, создание экземпляров классов и посылку последовательности сообщений экземплярам. Синтаксис Смолтока определяет три вида сообщений: унарные, бинарные и с ключевыми словами. Успешное использование языка требует от программиста знание всех основных видов объектов системы и сообщений которые им можно послать.

Семантика и синтаксис языка относительно просты. Тем не менее система большая и сложная из за количества видов доступных объектов. В системе Смолток есть восемь важных категорий классов: ядро и поддержка ядра, скалярные величины, числа, совокупности, потоки, классы, независимые процессы и графика. Протокол этих видов объектов рассматривается в 12 главах второй части. В каждой из этих глав приводится диаграмма ирархии классов данная в первой главе чтобы показать часть иерархии обсуждаемой в текущей главе. Две дополнительные главы во второй части показывают примеры предложений Смолтока и описаний классов. Классы в системе Смолток определяются в линейной иерархии. Главы второй части предоставляют энциклопедический обзор протокола классов: категории определённых сообщений, каждое сообщение прокомментировано и приведены примеры. Однако не смотря на присутствие протокола класса описаны только сообщения добавленные классом. Полный протокол сообщений определяется просмотром протокола определённого в классе и в каждом его надклассе. Удобно описывать классы начиная с класса *Объект* и продолжать описание следуя цепи наследования так чтобы унаследованный протокол можно было понимать вместе с новым протоколом.

Глава 6

Протокол для всех объектов

Оглавление

Всё что есть в системе это объекты. Протокол общий для всех объектов системы предоставлен в описании класса *Объект*. Это значит что любой объект созданный системой может отвечать на сообщения определённые в классе *Объект*. Обычно это сообщения которые поддерживают разумное поведение по умолчанию для предоставления начальной точки с которой можно продолжать создавать новые виды объектов, либо добавляя новые сообщеня либо изменяя ответ на существующие сообщения. Примерами для иллюстрации протокола *Объекта* являются объекты числа такие как 3 или 16.23, наборы такие как 'this is a string' или #(#это #ряд), *пусто* или *истина*, и объекты описывающие класс такие как *Набор* или *Малое целое* или, даже, сам *Объект*.

Описание протокола для класса *Объект* данное в данной главе

Объект

- Величина
  - Знак
  - Дата
  - Время
  - Число
    - Плавающее
    - Дробь
    - Целое
      - Большое положительное целое
      - Большое отрицательное целое
      - Малое целое
  - Ключ поиска
    - Ассоциация
- Связь
  - Процесс
- Набор
  - Набор последовательность
    - Связанный список
      - Семафор
    - Набор ряд
      - Ряд
        - Растровое изображение
        - Ряд серий
        - Цепь
          - Символ
        - Текст
        - Ряд байтов
    - Интервал
    - Упорядоченный набор
      - Сортированный набор
  - Мешок
  - Набор отображение
  - Множество
    - Словарь
      - Тождественный словарь
- Поток
  - Позиционируемый поток
    - Поток чтения
    - Поток записи
      - Поток чтения записи
        - Поток файл
  - Случайное число
- Неопределённый объект
- Логика
  - Истина
  - Ложь
- Планировщик исполнителя
- Задержка
- Разделяемая очередь
- Поведение
  - Описание класса
    - Класс
    - Метакласс
- Точка
- Прямоугольник

не полно. Опущены сообщения принадлежащие обработке сообщений, отношениям зависимости, и примитивам системы. Они представлены в главе 14.

6.1 Проверка функциональности объекта

Каждый объект это экземпляр класса. Функциональность объекта определяется его классом. Эта функциональность проверяется двумя способами: явное указание класса для определения того что это класс или надкласс объекта, и указание селекторов сообщений для определения может ли объект отвечать на них. Эти два способа отражают два способа мышления о взаимосвязи между экземплярами различных классов: в терминах ирархии классов/подклассов, или в терминах разделяемых протоколов сообщений.

Протокол экземпляров *Объекта*

принадлежность классу

класс
Возвращает объект являющийся классом получателя.

это разновидность: **класс**
Отвечает является ли аргумент, класс, надклассом или классом получателя.

это член: **класс**
Отвечает является ли получатель прямым экземпляром аргумента, класс. Это то же самое что проверить является ли ответ на сообщение класс тем же (==) что и аргумент класс.

отвечает на: **символ**
Отвечает содержит ли словарь класса получателя или его надкласса аргумент, символ, в качестве селектора сообщения.

Примеры сообщений и их результатов:

предложение	значение
3 класс.	*Малое целое*
#(#это #ряд) это разновидность: Набор.	*истина*
#(#это #ряд) это член: Набор.	*ложь*

#(#это #ряд) класс.	*Ряд*
3 отвечает на: #это разновидность:.	*истина*
#(1 2 3) это член: Ряд.	*истина*
Объект класс.	*Объект класс*

6.2 Сравнение объектов

Т.к. вся информация в системе представлена объектами, то существует основной протокол для проверки идентичности объектов и для копирования объектов. Важные методы сравнения определённые в классе *Объект* это тесты на эквивалентность и на равенство. Эквивалентность (==) это проверка являются ли два объект одним и тем же объектом. Равенство (=) это проверка на то представляют ли объекты одинаковые компоненты. Смысл фразы «представляют ли объекты одинаковые компоненты» зависит от получателя сообщения; каждый новый вид объектов который добавляет переменные экземпляра обычно должен переопределить сообщение = чтобы определить какие переменные экземпляра должны участвовать в определении равенства. Например, равенство двух рядов определяется при помощи проверки их размеров и затем проверки равенства каждого элемента массивов; равенство двух чисел определяется проверкой представляют ли числа одно и то же значение; равенство двух счётов в банке может быть проверено только проверкой равенства идентификационного номера.

Сообщение *хэш* это специальная часть протокола сравнения. Ответ на *хэш* это целое. Любые два равные объекта должны возвращать одно и то же значение хэша. Неравные объекты могут либо не могут возвращать равные значения хэша. Обычно это целое значение используется как номер для сохранения объекта в нумерованном наборе (как показано в главе 3). Всегда когда переопределяется сообщение = также должно быть переопределено сообщение *хэш* для сохранения свойства равенства хэшей у равных объектов.

Протокол экземпляров *Объекта*

сравнение

== объект
Отвечает являются ли получатель и аргумент одним и тем же объектом.

= объект
Отвечает представляют ли получатель и аргумент одинаковые компоненты.

~= объект
Отвечает представляют ли получатель и аргумент разные компоненты.

~~ объект
Отвечает являются ли получатель и аргумент разными объектами.

хэш
Возвращает целое вычисленное на основе значения получателя.

Реализация по умолчанию сообщения = та же что и для сообщения ==.

Некоторые специализированные протоколы сравнения предоставляют краткий способ проверить идентичность с объектом *пусто*.

Протокол экземпляров *Объекта*

проверки

это пусто
Отвечает является ли получатель нулём.

не пусто
Отвечает является ли получаетль отличным от нуля.

Эти сообщений соответветственно идентичны следующим предложениям == пусто и ~~ пусто. Выбор используемого способа зависит от персонального стиля программирования.

Некоторые очевидные примеры:

предложение	значение
пусто это пусто.	*истина*
истина не пусто.	*истина*

3 это пусто.	*ложь*
#(#а #б #в) = #(#а #б #в).	*истина*
3 = (6 / 2).	*истина*
#(1 2 3) класс == Ряд.	*истина*

6.3 Копирование объектов

Есть два способа сделать копию объекта. Разница заключается в том копируются или нет переменные объекта. Если переменные не копируются, то они разделяются (*поверхностная копия*); если значения переменных копируются, то они не разделяются между объектами (*глубокая копия*).

Протокол экземпляров *Объекта*

копирование

копия
Возвращает другой экземпляр такой же как получатель.

поверхностная копия
Возвращает копию получателя которая разделяет переменные экземпляра с исходным объектом.

глубокая копия
Возвращает копию получателя со своей собственной копией переменных экземпляра.

Реализация по умолчанию для метода *копия* это *поверхностная копия*. Метод *копия* обычно переопределяется в подклассах где результатом копирования является комбинация разделяемых и не разделяемых переменных, в отличии от методов *поверхностная копия* и *глубокая копия*.

Например копия (*поверхностная копия*) *Ряда* ссылается на те же элементы что и исходный *Ряд*, но копия это другой объект. Замена элемента в копии не изменяет исходного объекта.

предожение	результат
а ← #('первый' 'второй' 'третий').	('первый' 'второй' 'третий')
б ← а копия.	('первый' 'второй' 'третий')
а = б.	*истина*
а == б.	*ложь*
а от: 1 == (б от: 1).	*истина*
б от: 1 пом: 'новый первый'.	'новый первый'
а = б.	*ложь*
а ← 'привет'.	'привет'
б ← а копия.	'привет'
а = б.	*истина*
а == б.	*ложь*

Рисунок 6.1 показывает взаимоотношение между поверхностной и глубокой копией. Рассмотрим пример *Записи о человеке*, для

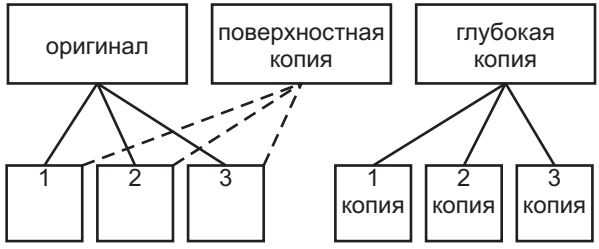


Рис. 6.1

дальнейшей иллюстрации разницы между поверхностной копией и глубокой копией. Допустим в определении записи указана переменная *страховка*, экземпляр класса *Страховка*. Предположим также что каждый экземпляр *Страховки* имеет значение связанное с объёмом медицинского обслуживания. Предположим что мы создали запись о служащем как экземпляр прототип *Записи о человеке*. Под словом прототип мы подразумеваем что объект имеет все значения как у любого нового экземпляра класса, поэтому этот экземпляр может быть создан просто копированием а не посылкой последовательности инициализирующих сообщений. Предположим также что этот экземпляр прототип является переменной класса *Запись о человеке* и что ответ на создание новой *Записи о человеке* это *поверхностная копия* этой переменной; поэтому метод связанный с сообщением *новый* это ↑ *запись о служащем копия*.

В результате выполнения предложения:

запись Иванов Иван ← Запись о человеке новый.

запись Иванов Иван ссылается на копию (в действительности на поверхностную копию) записи о служащем.

Прототип *запись о служащем* и *запись Иванов Иван* разделяют одну и ту же страховку. Политика компании может измениться. Допустим *Запись о человеке* понимает сообщение *изменить предел страховки: число*, которое реализуется при помощи изменения предела страховки записи о служащем. Т.к. эта страховка разделяется, то результатом выполнения предложения:

Запись о человеке изменить предел страховки: 4000.

будет изменение объёма медицинских услуг для всех работников. В примере изменится и страховка записи о служащем и страховка записи Иванов Иван. Сообщение *изменить предел страховки:* посылается классу *Запись о человеке* т.к. это подходящий объект для сбора сведений об изменении всех экземпляров.

6.4 Доступ к частям объекта

В системе Смолток есть два вида объектов, объекты с именованными переменными и объекты с нумерованными переменными. Объекты с нумерованными переменными могут также иметь именованные переменные. Это различие было рассмотрено в главе 3. Класс *Объект* поддерживает шесть сообщений предназначенных для доступа к нумерованным переменным объекта. Вот эти сообщения:

Протокол экземпляров *Объекта*

доступ

от: номер
Возвращает значение нумерованной переменной экземпляра получателя чей номер это аргумент, номер. Если получатель не имеет нумерованных переменных, или аргумент больше чем количество нумерованных переменных то выдаётся сообщение об ошибке.

от: номер пом: объект
Помещает аргумент, объект, как значение нумерованной переменной экземпляра получателя чей номер это аргумент, номер. Если получатель не имеет нумерованных переменных, то выдаётся сообщение об ошибке. Возвращает объект.

основной от: номер
То же что и *от: номер*. Однако метод связанный с этим сообщением не может быть изменён в любом подклассе.

основной от: номер пом: объект
То же что и *от: номер пом: объект*. Однако метод связанный с этим сообщением не может быть изменён в любом подклассе.

размер

Возвращает количество доступных нумерованных переменных. Это то же самое что и наибольший допустимый номер.

основной размер
То же что и *размер*. Однако метод связанный с этим сообщением не может быть изменён в любом подклассе.

Заметьте что метод доступа идут парами, одно сообщение в каждой паре начинается со слова основной означающим что это базисное сообщение системы чья реализация не должна изменяться в любом подклассе. Причина предоставления этих пар сообщений в том что внешний протокол; *от:*, *от:пом:* и *размер*; может быть переопределён для специальных случаев, в то же время остаётся способ получить доступ к примитивным методам. (в главе 4 приведено объяснение примитивных методов, это методы которые реализуются для системы виртуальной машиной.) Поэтому в любом методе ирархии описания класса сообщение *основной от:*, *осовной от:пом:* и *основной размер* должны всегда использоваться для доступа к примитивным методам. Сообщение *основной размер* может быть послано любому объекту; если объект не переменной длины, то ответом будет 0.

Экземпляры класса *Ряд* это объекты с переменной длиной. Допустим *буквы* это *Ряд* #($а $б $в $г $д), тогда:

предложение	результат
буквы размер.	5
буквы от: 3.	$в
буквы от: 3 пом: $е.	$е
буквы.	#($а $б $е $г $д)

6.5 Печать и сохранение объектов

Есть много способов создать последовательность знаков которая предоставляет описание объекта. Описание может быть только ключём например для отличия объекта, или описание может быть достаточно подробным чтобы можно было создать подобный объект. В первом случае (печать), описание может быть хорошо форматирова-

но, например как результат функции Lisp pretty-printing. Во втором случае (сохранение), описание может сохранять информацию разделяемую с другими объектами.

Протокол сообщений классов системы Смолток поддерживает печать и сохранение. Реализация этих сообщений в классе *Объект* предоставляет минимальные возможности; большинство классов переопределяют сообщения чтобы улучшить создаваемые описания. Аргументы двух сообщений это экземпляры разновидности *Потока*; Потоки описаны в главе 12.

Протокол экземпляров *Объекта*

печать

цепь для печати
Возвращает Цепь чьи знаки описывают получателя.

печать в: **поток**
Добавляет к аргументу, поток, цепь чьи знаки описывают получателя.

сохранение

цепь для сохранения
Возвращает цепь представляющую получателя из которой он может быть воссоздан.

сохранить в: **поток**
Добавляет к аргументу, поток, цепь представляющую получателя из которой он может быть воссоздан.

Каждый из двух видов печати основан на создании последовательности знаков которые могут быть показаны на мониторе, записаны в файл или переданы по сети. Последовательность созданная сообщениями *цепь для сохранения* или *сохранить в:* должна быть проинтерпретирована как одно или более предложений которые могут быть выполнены чтобы воссоздать объект. Например *Множество* из трёх элементов $а $б $в может быть напечатано как
Множество ($а $б $в).
в то время как оно может быть сохранено как
(Множество новый добавить: $а; добавить: $б; добавить: $в.)

Литералы могут использовать одно и то же представление для печати и для сохранения. Поэтому цепь 'привет' должна печататься и сохраняться как 'привет'. Символ #имя печатается как имя, но сохраняется как #имя.

Для сохранения наибольшей информации реализация по умолчанию для *цепи для печати* это имя класса объекта; реализация по умолчанию для *цепь для сохранения* это имя класса со следующим сообщением создания экземпляра *основной новый* и последующая последовательность сообщений для сохранения переменных экземпляра. Например если подкласс *Объекта*, скажем класс *Пример*, демонстрирует поведение по умолчанию, тогда для экземпляра *Примера* с именем *прим* у которого нет переменных экземпляра, мы должны получить:

предложение	**результат**
прим цепь для печати.	Пример
прим цепь для сохранения.	'(Пример основной новый)'

6.6 Обработка ошибок

Тот факт что все действия выполняются при помощи посылки сообщений объектам означает что существует только одна причина ошибки которая должна обрабатываться системой: сообщение послано объекту, но сообщение не определено ни в одном классе цепи наследования. Эта ошибка обнаруживается интерпретатором чьей реакцией является посылка исходному объекту сообщения *не понимаю: сообщение*. Аргумент, *сообщение*, представляет селектор неудачного сообщения и связанные с ним аргументы, если они есть. Метод связанный с селектором *не понимаю:* выдаёт для пользователя сообщение что произошла ошибка. Способ представления этого сообщения пользователю является функцией (графического) интерфейса поддерживаемая системой и не описывается здесь; минимальным требованием интерактивной системы является возможность напечатать сообщение об ошибке на устройсто вывода и затем дать пользователю возможность исправить ошибочную ситуацию. Глава

17 описывает сообщение об ошибке системы Смолток и методы отладки.

В дополнение к основному сообщению об ошибке, методы могут явно использовать механизм обработки ошибок системы для случаев в которых проверка определяет что программа совершает что то недопустимое. В таком случае метод может задать описание ошибки которое будет показано пользователю. Обычный способ сделать это — послать текущему экземпляру сообщение *ошибка: цепъ*, где аргумент представляет желаемый комментарий. Реализация по умолчанию для этого сообщения — вызов механизма оповещения системы. Программист может предоставить альтернативную реализацию для *ошибки:* которая использует зависящее от программы сообщение об ошибке.

Общие сообщения поддерживаются в протоколе класса *Объект*. Сообщение об ошибке может указывать что примитив системы не выполнился или что подкласс переопределил унаследованное сообщение которое не поддерживается и что пользователь не должен его вызывать или что надкласс определяет сообщения которые должны быть реализованы в подклассах.

Протокол экземпляров *Объекта*

обработка ошибок

не понимаю: **сообщение**
Сообщает пользователю что получатель не понимает аргумента, сообщение, в качестве сообщения.

ошибка: **цепь**
Сообщает пользователю что произошла ошибка при ответе на сообщение посланное получателю. Сообщение использует аргумент, цепь, как часть уведомления об ошибке.

примитив не выполнен
Сообщает пользователю что метод реализованый как примитив системы не смог правильно завершиться.

не должен реализовывать

Сообщает пользователю что не смотря на то что надкласс получателя определяет это сообщение оно не должно быть реализовано в подклассе, класс получателя не может предоставить подходящей реализации.

ответственность подкласса
Сообщает пользователю что метод определённый в надклассе получателя должен быть реализован в классе получателя.

Подклассы могут переопределять сообщения обработки ошибок чтобы предоставить специальную поддержку для соответствующих ошибочных ситуаций. Глава 13 которая описывает реализацию классов наборов даёт примеры использования последних двух сообщений.

Глава 7

Скалярные величины

Оглавление

Система Смолток предоставляет несколько классов представляющих объекты которые измеряют что либо с линейным порядком. Примеры таких измеряемых количеств из реального мира это (1) временные отрезки такие как даты и время, (2) пространственные количества такие как расстояние и (3) численные значения такие как действительные и рациональные числа.

7.1 Класс *Величина*

Является ли одно число меньше другого? Следует ли одна дата после другой даты? Предшествует ли одно время другому времени? Следует ли буква после другой в алфавите? Данное расстояние такое же или меньше чем друго расстояние?

Общий протокол для ответа на данные вопросы предоставляется классом *Величина*. *Величина* предоставляет протокол для объектов которым нужно иметь возможность сравниваться с одномер-

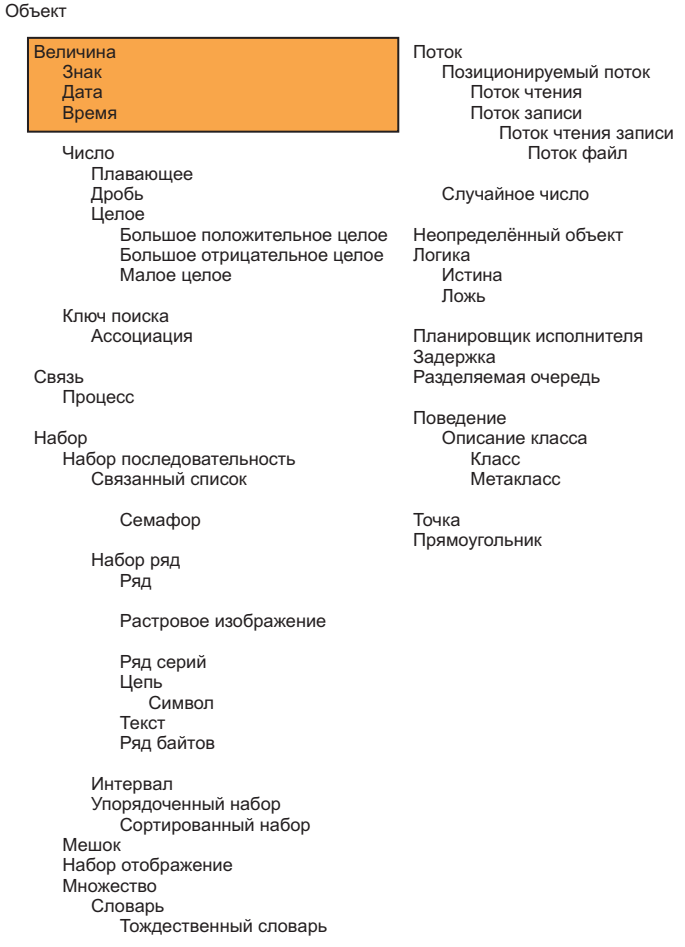

Объект
Величина
Знак
Дата
Время
Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое
Ключ поиска
Ассоциация
Связь
Процесс
Набор
Набор последовательность
Связанный список
Семафор
Набор ряд
Ряд
Растровое изображение
Ряд серий
Цепь
Символ
Текст
Ряд байтов
Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь
Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл
Случайное число
Неопределённый объект
Логика
Истина
Ложь
Планировщик исполнителя
Задержка
Разделяемая очередь
Поведение
Описание класса
Класс
Метакласс
Точка
Прямоугольник

Рис. 7.1

ными величинами. Подклассами класса *Величина* являются *Дата*, *Время* и *Число*. Классы *Знак* (элемент цепи) и *Ключ поиска* (ключ в ассоциации словаря) также реализованы как подклассы класса *Величина*. *Знаки* интересны как пример неизменных объектов системы и поэтому они введены в данной главе; *Ключ поиска* это менее интересный объект и его рассмотрение отложено до тех пор пока мы не станем изучать главу о наборах. Класса *Расстояние* нет в текущей версии системы Смолток.

Протокол экземпляров *Величины*

сравнение

< величина
: Отвечает является ли получатель меньше аргумента.

<= величина
: Отвечает является ли получатель меньше чем или равен аргументу.

> величина
: Отвечает является ли получатель больше аргумента.

>= величина
: Отвечает является ли получатель больше чем или равен аргументу.

между: мин и: макс
: Отвечает лежит ли получатель в интервале между аргументами мин и макс, включая границы. Т.е. он вычисляет условие сам >= мин и сам <= макс.

Несмотря на то что *Величина* наследует от своего надкласса, *Объекта*, сообщение = для сравнения на равенство двух объектов, каждый вид *Величины* должен переопределить это сообщение. Метод связанный с селектором = в классе *Величина*:

```
сам ответственность подкласса.
```

Если подкласс *Величины* не реализовал =, то при попытке послать сообщение экземпляру подкласса появится сообщение об ошибке что подкласс должен реализовать это сообщение, как указано в его надклассе.

Экземпляры видов *Величины* также могут отвечать на сообщения которые определяют какой из двух объектов больше или меньше.

Протокол экземпляров *Величины*

проверки

мин: **величина**
Возвращает получателя или аргумент, то у чего меньше величина.

макс: **величина**
Возвращает получателя или аргумент, то у чего больше величина.

Заметьте что протокол для идентичности ==, ~= и ~~ наследуется от класса Объект. Используя целые числа в качестве примера вида *Величины* получаем:

предложение	результат
3 <= 4.	*истина*
3 > 4.	*ложь*
5 между: 2 и: 6.	*истина*
5 между: 2 и: 4.	*ложь*
34 мин: 45.	34
34 макс: 45.	45

Программист не может создавать экземпляры *Величины*, а только её подклассы. Это происходит из за того что *Величина* не реализует все сообщения которые объявляет, т.к. она реализует несколько своих сообщений при помощи предложения *сам ответственность подкласса*.

7.2 Класс *Дата*

Сейчас, после рассмотрения основных сообщений протокола *Величины*, можно добавить дополнительный протокол для поддержки

арифметики и специальных запросов о линейный величинах. Первое уточнение которое мы рассмотрим будет подкласс *Дата*.

Экземпляры *Даты* представляют день от начала Юлианского календаря. День существует в виде конкретного месяца и года. Класс *Дата* знает о некоторых очевидных вещах: (1) в неделе 7 дней, каждый день имеет имя и номер 1, 2, . . . , 7; (2) в году 12 месяцев, каждый имеет имя и номер 1, 2, . . . , 12; (3) в месяце 28, 29, 30 или 31 день; и (4) некоторые годы являются високосными.

Протокол предоставляемый для объекта *Дата* поддерживает запросы о *Датах* в общем и о конкретных *Датах*. И *Дата* и *Время* являются интересными примерами классов системы у которых есть специальные свойства доступные для самих классов, в отличии от их экземпляров. Этот «протокол класса» определён в метаклассе класса. Давайте сначала рассмотрим протокол класса *Дата* который поддерживает общие запросы.

Протокол класса *Дата*

общие запросы

день недели: имя дня
Возвращает номер дня в неделе, 1, 2, . . . , 7 переданного в качестве аргумента, имя дня.

имя дня: номер дня
Возвращет символ который представляет имя дня чей номер это аргумент, номер дня, где 1 — понедельник, 2 — вторник, и т.д.

номер месяца: имя месяца
Возвращает номер месяца в году, 1, 2, . . . , 12 переданного в качестве аргумента, имя месяца.

имя месяца: номер месяца
Возвращает символ который представляет имя месяца с номером являющимся переданным аргументом, номер месяца, где 1 — январь, 2 — февраль, и т.д.

дней в месяце: имя месяца **для года:** число год
Возвращает количество дней в месяце чьё имя это имя месяца в году число года (год нужен для подсчёта дней в високосном году).

дней в году: число год
Возвращает число дней в году, число год.

високосный год: число год
Возвращает 1 если число год это високосный год; иначе возвращает 0.

текущие дата и время
Возвращает ряд чей первый элемент это текущая дата (экземпляр класса *Дата* представляющий текущую дату) а второй элемент это текущее время (экземпляр класса *Время* представляющий текущее время).

Поэтому можно посылать следующие сообщения:

предолжение	результат
Дата дней в году: 1982.	365
Дата день недели: #Среда.	3
Дата имя месяца: 10.	Октябрь

Дата високосный год: 1972.	1 (означает что год високосный)
Дата дней в месяце: #Февраль для года: 1972.	29
Дата дней в месяце: #Февраль для года: 1971.	28

Класс *Дата* знает сокращения для имён месяцев.

Есть четыре сообщения которые могут быть использованы для создания экземпляров класса *Дата*. Один широко используется в системе Смолток, особенно для создания даты файла, это Дата сегодня.

Протокол класса *Дата*

создание экземпляров

сегодня
Возвращает экземпляр *Даты* представляющей день когда послано сообщение.

из дней: количество дней
Возвращает экземпляр *Даты* которая является днём номер количество дней после или до 1 января 1901 года (в зависимости от знака аргумента).

новый день: день месяц: имя месяца год: число год
Возвращает экземпляр *Даты* являющейся днём номер день в месяце именуемом имя месяца в году число год.

новый день: количество дней год: число год
Возвращает экземпляр *Даты* которая является днём номер количество дней от начала года число год.

Четыре примера сообщений создания экземпляра:

предложение	**результат**
Дата сегодня.	3 февраля 1982
Дата из дней: 200.	20 июля 1901
Дата новый день: 6 месяц: #фев год: 82.	6 февраля 1982
Дата новый день: 3 год: 82.	3 января 1982

Сообщения которые можно послать экземплярам *Даты* разделены на следующие категории: доступ, запросы, арифметика и печать. Категории доступ и запросы содержат:

- номер дня, номер месяца или год
- количество секунд, дней или месяцев с момента другой даты
- общее количество дней в месяце или годе даты
- количество дней оставшихся в месяце или годе даты
- первый день месяца даты
- имя дня недели или месяца даты
- дата некоторого дня недели предшествующего экземпляру

Протоколом класса *Дата* поддерживается простая арифметика.

Протокол экземпляров *Даты*

арифметика

добавить дни: **количество дней**
Возвращает *Дату* на количество дней позже чем получатель.

вычесть дни: **количество дней**
Возвращает *Дату* которая на количество дней раньше чем получатель.

вычесть дату: **дата**
Возвращает *Целое* которое представляет количество дней между получателем и аргументом, дата.

Данная арифметика полезна, например, чтобы вычислить дату возврата для книги из библиотеки или плату за просроченную книгу. Допустим дата возврата это экземпляр *Даты* указывающий день до которого книга должна быть возвращена в библиотеку. Тогда:

Дата сегодня вычесть дату: дата возврата.

вычисляет количество дней за которые нужно заплатить пеню. Если книга взята сегодня на две недели, тогда:

```
Дата сегодня добавить дни: 14.
```

это дата возврата книги. Если библиотека заканчивает роботать за 16 дней до Рождества, то дата последнего рабочего дня это:

```
(Дата новый день: 25 месяц: #December год: 1982) вычесть дни: 16.
```

Алгоритм вычисляющий размер пени которую должен заплатить должник сначала сравнивает текущую дату с датой возврата и затем, если дата возврата уже прошла, вычисляет пеню как 10 центов умноженных на количество дней превышения над датой возврата.

```
Дата сегодня < дата возврата
    истина: [ пеня ← 0. ]
    ложь: [ пеня ← 0.10 * (Дата сегодня вычесть дату: дата возврата). ].
```

7.3 Класс *Время*

Экземпляры класса *Время* представляют некоторую секунду в дне. День начинается в полночь. *Время* это подкласс *Величины*. Подобно классу *Дата*, *Время* может отвечать на общие сообщеня запросы которые определены в протоколе класса.

Протокол класса *Время*

общие запросы

значение часов в миллисекундах
Возвращает количество миллисекунд с момента когда когда часы были сброшены или перешли через ноль.

миллисекунд на выполнение: замеряемый блок
Возвращает количество миллисекунд потребовавшихся на выполнение аргумента, замеряемый блок.

всего секунд
Возвращает общее количество секунд от 1 января 1901 года с поправкой на временную зону и переводом часов на летнее время.

текущие дата и время

Возвращает ряд чей первый элемент это текущая дата (экземпляра класса *Дата* который представляет текущую дату) и второй элемент это текущее время (экземпляр класса *Время* который представляет текущее время). Результат посылки этого сообщения *Времени* идентичен результату посылке его *Дате*.

Единственный не очевидный запрос это *миллисекунд на выполнение: замеряемый блок*. В примере:

```
Время миллисекунд на выполнение: [ Дата сегодня. ].
```

результатом будет количество миллисекунд потребовавшихся системе для вычисления сегодняшней даты. Из за того что существуют некоторые накладные расходы при ответе на это сообщение, и из за того что разрешение часов влияет на результат, то осторожный программист должен определять машинно-зависимую неточность связанную с выбором подходящего аргумента для этого сообщения.

Новый экземпляр *Времени* можно создать послав *Времени* сообщение сейчас; соответствующий метод читает текущее время из часов системы. В качестве альтернативы экземпляр *Времени* можно создать послав сообщение *из секунд: количество секунд*, где количество секунд это число секунд от полуночи.

Протокол класса *Время*

создание экземпляров

текущее
Возвращает экземпляр *Времени* представляющий секунду посылки сообщения.

из секунд: количество секунд
Возвращает экземпляр *Времени* который является количеством секунд от полуночи.

Протокол доступа для экземпляров *Времени* предоставляет такую информацию как количество часов (*часы*), минут (*минуты*) и секунд (*секунды*) которые представляет экземпляр. Так же поддерживается арифметика.

Протокол экземпляров *Времени*

арифметика

добавить время: **количество времени**
Возвращает экземпляр *Времени* который на аргумент, количество времени, больше чем получатель.

вычесть время: **количество времени**
Возвращает экземпляр *Времени* который на аргумент, количество времени, меньше чем получатель.

В выше приведённых сообщениях аргумент (количество времени) может быть или *Датой* или *Временем*. Чтобы это было возможным система должна уметь переводить *Дату* и *Время* в общую единицу измерения, она их переводит в секунды. В случае *Времени*, переведённая величина это количество секунд от полуночи, в случае *Даты*, переведённая величина это количество секунд между 1 января 1901 года и тем же временем в дне получателя. Для поддержки этих методов, экземпляры каждого класса отвечают на преобразующий метод *как секунды*.

Протокол экземпляров *Времени*

преобразование

как секунды
Возвращает количество секунд от полуночи которое представляет получатель.

Протокол экземпляров *Даты*

преобразование

как секунды
Возвращает число секунд между временем 1 января 1901 года и тем же временем в дне представляемом получателем.

Арифметика для *Времени* может быть использована способами аналогичными способам для *Даты*. Допустим количество времени

потраченное человеком на работу над некоторым проектом должно заносится в журнал, чтобы заказчик мог изменять почасовую оплату. Допустим человек начал работу во время начала и работал непрерывно в течении дня до данного момента; зазвонил телефон — заказчик хочет узнать сегодняшние расходы. В этот момент оплата составляет 5 долларов в час:

(Время текущее вычесть время: время начала) часы * 5.

не учитывает долполнительные секунды или минуты. Если оплата любой части часа большей 30 минут должна быть как за полный час тогда если:

(Время текущее вычесть время: время начала) минуты > 30.

добавляются дополнительные 5 долларов.

Кто более продуктивен, работник который закончил работу с записью в журнал во время а или работник с временем б? Ответ — первый работник если время а < времени б. Протокол сравнения наследуется от надклассов *Величина* и *Объект*.

Допустим времена вычисляются в течении дня, например, при вычислении времени машины в четырёхдневном ралли. Если в первый день ралли началось во время начала дня *день начала*, тогда время прибытия машины на финишную линию вычисляется следующим образом. Допустим что время старта 6:00.

время начала ← Время из секунд: 60 * 60 * 6.

2 февраля 1982 года.

дата старта ← Дата новый день: 2 месяц: #фев год: 82.

Время прошедшее до начала текущего дня это:

начало сегодня ← (((Время из секунд: 0) добавить время: Дата сегодня) вычесть время: дата старта)
вычесть время: время начала.

Здесь добавляются все секунды от начала 1 января 1901 года до начала сегодняшнего дня и затем вычитаются все секунды от начала 1 января 1901 года до начала даты старта. Это эквивалентно добавлению количество секунд в количестве дней гонки, но затем программист должен произвести все преобразования.

(Дата сегодня вычесть дату: дата старта) * 24 * 60 * 60.

Добавив текущее время мы получим время потраченное машиной на прохождение ралли.

начало сегодня добавить время: Время текущее.

7.4 Класс *Знак*

Класс *Знак* это третий подкласс класса *Величина* который мы рассмотрим. Это вид *Величины* т.к. экземпляры класса *Знак* принадлежат упорядоченной последовательности и им можно задавать вопрос предшествует ли данный знак (<) или следует после (>) другого знака в алфавите.

Знаки могут быть записаны при помощи литерала с помощью предшествующего знака доллара ($); так $A это *Знак* представляющий большую букву «А». Протокол для создания экземпляров *Знака* состоит из:

Протокол класса *Знак*

создание экземпляров

значение: **целое**
Возвращает экземпляр *Знака* чьё значение это аргумент, целое. Значение это code point уникод символа. Например, *Знак значение: 16o1040* это большая буква «А».

значение цифры: **целое**
Возвращет экземпляр *Знака* который является цифрой с номером равным аргументу, целое. Например ответ будет $9 если аргумент это 9; ответ $0 если аргумент 0; ответ $A если аргумент 10, и ответ $Z если аргумент равен 35. Этот метод полезен при преобразвании чисел в цепи. Обычно используются буквы только до $F (для шестнадцатиричных чисел).

Протокол класса содержит набор сообщений для доступа к знакам которые сложно различать по внешнему виду: *забой*, *пс* (перевод строки), *эскейп*, *новая страница*, *пробел* и *таб* (табуляция).

Сообщения экземпляров *Знака* поддерживают доступ к значению кода знака и проверку типа знака. Единственное состояние *Знака* это его значение которое никогда не может быть изменено. Объекты которые не изменяют своё состояние называются неизменными объектами. Это означает что будучи однажды созданы они не разрушаются и воссоздаются когда они снова становятся нужными. Когда создаётся новый *Знак* с кодом между 0 и 255 возвращается ссылка на уже существующий *Знак*. Поэтому 256 *Знаков* уникальны. Кроме *Знаков* система Смолток содержит *Малые целые* и *Символы* которые являются неизменными объектами.

Протокол экземпляров *Знака*

доступ

значение АСКОИ
Возвращает число соответствующее кодирове АСКОИ получателя (американский стандартный код обмена информацией).

значение цифры
Возвращает число соответствующее цифре системы счисления представляемой получателем (см. сообщения создания экземпляра *значение цифры:*).

проверки

это буква или цифра
Возвращает истину если получатель это буква или цифра.

это цифра
Отвечает является ли получатель цифрой.

это буква
Отвечает является ли получатель буквой.

в нижнем регистре
Отвечает является ли получатель буквой в нижнем регистре.

в верхнем регистре
Отвечает является ли получатель буквой в верхнем регистре.

это разделитель
Отвечает является ли получатель одним из знаков разделителей в синтаксисе предложений: пробел, пс, таб, перевод строки или перевод страницы.

это гласный
Отвечает является ли получатель одним из гласных: a, e, i, o или u в верхнем или нижнем регистре.

Протокол экземпляра также предоставляет протокол преобразования букв в верхний или нижний регистр (*в верхнем регистре* и *в нижнем регистре*) и преобразования в символ (*как символ*).

Простое сравнение по алфавиту показывает использование протокола сравнения для экземпляров *Знака*. Допустим мы хотим узнать прешествует ли одна цепь другой в телефонной книге. *Цепь* предоставляет сообщение *от:* для нахождения элемента с номером переданным в качестве аргумента; элементы *Цепей* это *Знаки*. Поэтому *'абв' от: 2* это $б. Ниже подразумевается что в классе *Цепь* определён метод *мин:*. Метод возвращает *Цепь*, получателя сообщения или его аргумент, то что идёт первым в алфавитном порядке.

```
мин: цепь
  1
    до: сам размер
    делать: [
      :номер |
      номер > цепь размер истина: [ ↑ цепь. ].
      (сам от: номер) > (цепь от: номер) истина: [ ↑ цепь. ].
      (сам от: номер) < (цепь от: номер) истина: [ ↑ сам. ]. ].
  ↑ сам.
```

Алгоритм содержит два предложения. Первое это цикл по всем элементам получателя. Цикл заканчивается когда либо (1) аргумент, цепь, не имеет знака с которым можно сравнить следующий знак получателя (т.е. *номер* $>$ *цепь размер*); (2) следующий знак получателя следует за знаком в цепи (т.е. (*сам от: номер)* $>$ *(цепь от: номер)*); или (3) следующий знак получател идёт после следующего знака цепи. Для примера случая (1) сравним 'абв' и 'абвг' это сравнение заканчивается когда номер $= 4$; ответ это 'абв' она идёт раньше в алфавитном порядке. Для примера случая (2) сравним 'абфд' и 'абвг'. Когда *номер* $= 3$ $ф $>$ $в это истина; ответом будет 'абвг'. Пример случая (3), сравним 'ая' и 'бю' метод завершается когда номер $= 1$; ответ это 'ая'. В случае когда у получателя меньше знаков чем у аргумента, даже если получатель это начальная подцепь аргу-

мента, выполнение первого предложения заканчивается; результат это получатель. Примером могут быть 'абв' и 'абвг'.

Заметьте что арифметика для знаков не поддерживается. Например следующее предложение неправильно.

a ← $A + 1.

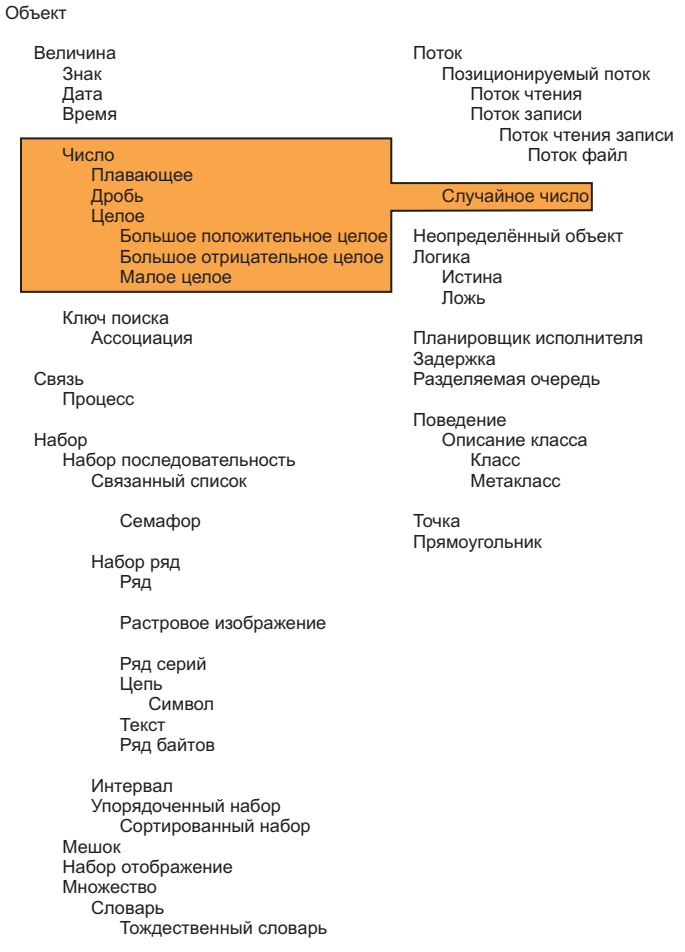
Возникнет ошибка т.к. знаки не понимают сообщения +.

Глава 8

Классы чисел

Оглавление

Одной из главных задач системы программирования Смолток является применение одной и той же метафоры для обработки информации настолько единообразно насколько это возможно. Метафора Смолтока, как было описано в предыдущих главах, это объекты которые взаимодействуют при помощи сообщений. Эта метафора очень похожа на метафору используемую в Симуле для реализации моделирования систем. Одной из наибольших проблем применения метафоры Смолтока ко всем аспектам системы программирования была арифметика. Симула использует метафору объектов/сообщений только для высокоуровневого взаимодействия в реализации модели. Для арифметики, как и для большинства алгоритмических управляющих структур, Симула полагается на встроенный язык программирования Алгол с его реализацией чисел, операторов и синтаксиса. Точка зрения утверждающая что сложение двух целых должно рассматриваться как посылка сообщения встретила определённое сопротивление на ранней стадии развития Смолтока. Опыт показал что преимущества такой единообразности в язы-


Объект
Величина
Знак
Дата
Время
Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое
Случайное число
Ключ поиска
Ассоциация
Связь
Процесс
Набор
Набор последовательность
Связанный список
Семафор
Набор ряд
Ряд
Растровое изображение
Ряд серий
Цепь
Символ
Текст
Ряд байтов
Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь
Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл
Неопределённый объект
Логика
Истина
Ложь
Планировщик исполнителя
Задержка
Разделяемая очередь
Поведение
Описание класса
Класс
Метакласс
Точка
Прямоугольник

ке программирования перевешивает любые неудобства при реализации. Через несколько версий Смолтока была применена техника реализации уменьшающая накладные расходы на посылку сообщения для наиболее частых арифметических операций, поэтому цена единообразности не слишком высока.

Объекты представляющие числовые значения используются в большинстве систем Смолтока (как и в большинстве других языков программирования). Естественно числа используются для выполнения математических вычислений; они также используются в алгоритмах как номера, счётчики и обозначают состояния или условия (часто называемые флагами). Целые числа также используются как наборы битов над которыми выполняются логические операции.

Каждый вид числовых значений представлен классом. Классы чисел реализуются так чтобы все числа имели наиболее общий тип. Класс некоторого объекта числа определяется тем насколько нужна полная общность для представления его значения. Поэтому внешний протокол для всех объектов чисел наследуются от класса *Число*. У *Числа* есть три подкласса: *Плавающее*, *Дробъ* и *Целое*. Целое имеет три подкласса: *Малое целое*, *Большое положительное целое* и *Большое отрицательное целое*. Целые числа предоставляют дополнительный протокол для интерпретации чисел как последовательности битов. Этот протокол объявляется в классе *Целое*. Числа в системе это экземпляры классов *Плавающее*, *Дробъ*, *Малое целое*, *Большое положительное целое* и *Большое отрицательное целое*. Классы *Число* и *Целое* объявляют общий протокол, но они не определяют представление конкретных числовых значений. Поэтому не создаётся экземпляров *Числа* и *Целого*.

В отличии от других объектов которые могут изменять своё внутреннее состояние, единственное состояние числа это его значение, которое никогда не изменяется. Объект 3, например, никогда не должен изменить своё состояние на 4, иначе могут наступить пагубные последствия.

8.1 Протокол классов чисел

Число определяет протокол для всех классов чисел. Его сообщения поддерживают стандартные арифметические операции и срав-

нения. Большинство из этого должно быть реализовано подклассами *Числа* т.к. эти операции зависят от вида представления значения.

Протокол арифметических сообщений содержит обычные бинарные операции такие как +, −, * и / и несколько унарных сообщений и сообщений с ключевыми словами для вычисления модуля числа, изменения знака числа или целочисленного деления или нахождения остатка. Категории для арифметических сообщений следующие:

Протокол экземпляров *Числа*

арифметика

+ число
Возвращает сумму получателя и аргумента, число.

− число
Возвращает разность между получателем и аргументом, число.

*** число**
Возвращает результат умножения получателя на аргумент, число.

/ число
Возвращает результат деления получателя на аргумент, число. Заметьте что из за того что сохранятся наибольшая возможная точность, то если деление происходит с остатком, то результат будет экземляром *Дроби*.

// число
Возвращает целое определённое как частное с округлением к минус бесконечности.

\\ число
Возвращает целый остаток от деления с округлением к минус бесконечности. Это деление по модулю.

модуль
Возвращает значение модуля получателя.

минус
Возвращает число со знаком противоположным знаку получателя.

дел: число
Возвращает целое определяемое как целочисленное деление с округлением к нулю.

ост: число
Возвращает остаток от целочисленного деления с округлением к нулю.

обратное
Возвращает 1 делённую на получателя. Если получатель равен 0 то сообщается об ошибке.

Несколько примеров.

предложение	результат
1 + 10.	11
5,6 − 3.	2,6
5 − 2,6.	2,4
-4 модуль.	4
6 / 2.	3
7 / 2.	(7/2), дробь с числителем 7 и знаменателем 2
7 обратное.	(1/7), дробь с числителем 1 и знаменателем 7

Арифметические сообщения которые возвращают результат целочисленного деления и его остатка следуют двум соглашениям. По одному соглашению округление производится к нулю, по другому к минус бесконечности. Это даёт одинаковые результаты для положительных результатов т.к. ноль и минус бесконечность находятся в одном и том же направлении. Для отрицательных результатов, эти операции округляют результат в разных направлениях. Протокол *Чисел* предоставляет оба варианта.

Следующая таблица показывает взаимоотношение между селекторами.

результат	округление к минус бесконечности	округление к нулю
частное	//	дел:
остаток	\\	ост:

Примеры:

предложение	результат
6 дел: 2	3
7 дел: 2	3
(7 дел: 2) + 1	4
7 дел: 2 + 1	2
7 ост: 2	1
7 // 2	3
7 \\ 2	1

7 \\ 2 + 1	2
-7 дел: 2	-3
-7 ост: 2	-1
-7 // 2	-4
-7 \\ 2	1

Результат возвращаемый *дел:*, *ост:* или // всегда положительный если получатель и аргумент имеют одинаковые знаки, и отрицательный если их знаки различаются. \\ всегда возвращает положительное значение.

Дополнительные математические функции:

Протокол экземпляров *Числа*

математические функции

эксп
Возвращает плавающее число равное экспоненте в степени получателя.

лн
Возвращает натуральный логарифм получателя.

лог: число
Возвращает логарифм получателя по основанию число.

пол лог: основание
Возвращает пол от логарифма получателя с основанием, основание, где пол это ближайшее целое со стороны минус бесконечности.

в степени: число
Возвращает получателя возведённого в степень аргумента, число.

в целоей степени: целое
Возвращает получателя возведённого в степень аргумента, целое, где аргумент должен быть видом *Целого*.

квадратный корень
Возвращает плавающее число равное квадратному корню получателя.

в квадрате
Возвращает получателя умноженного на себя.

Примеры:

предложение	результат
2,718284 лн.	1,0
6 эксп.	403,429
2 эксп.	7,38906
7,38906 лн.	1,99998
2 лог: 2.	1,0
2 пол лог: 2.	1
6 лог: 2.	2,58496
6 пол лог: 2.	2
6 в степени: 1,2.	8,58579
6 в целой степени: 2.	36
64 квадратный корень.	8,0
8 в квадрате.	64

Свойства чисел такие как чётность или нечётность и положительность и отрицательность можно проверить при помощи следующих сообщений.

Протокол экземпляров *Числа*

проверки

чётное
: Отвечает является ли получатель чётным числом.

нечётное
: Отвечает является ли получатель нечётным числом.

отрицательное
: Отвечает меньше ли 0 получатель.

положительное
: Отвечает больше ли либо равен 0 получатель.

строго положительное
: Отвечает больше ли 0 получатель.

знак

Возвращает 1 если получатель больше 0, -1 если получатель меньше 0, иначе 0.

Свойства чисел которые связаны с округлением и усечением предоставляются следующим протоколом.

Протокол экземпляров *Числа*

усечение и округление

потолок
Возвращает целое ближайшее к получателю со стороны плюс бесконечности.

пол
Возвращает целое ближайшее к получателю со стороны минус бесконечности.

усечь
Возвращает целое ближайшее к получателю со стороны нуля.

усечь до: число
Возвращает ближайшее число кратное аргументу, число, со стороны нуля.

округлить
Возвращает ближайшее к получателю целое.

округлить до: число
Возвращает ближайшее к получателю число кратное аргументу, число.

Когда *Число* нужно преобразовать в *Целое*, то можно использовать сообщение усечённый. Таким образом получаем:

предложение	результат
16,32 потолок.	17
16,32 пол.	16
-16,32 пол.	-17
-16,32 усечь.	-16
16,32 усечь.	16
16,32 усечь до: 5.	15
16,32 усечь до: 5,1.	15,3
16,32 округлить.	16
16,32 округлить до: 6.	18
16,32 округлить до: 6,3.	18,9

Протокол предоставляемый классом *Число* включает различные сообщения для преобразования чисел к другому виду объекта или к

другой единице измерения. *Числа* могут представляться в различных единицах измерения таких как градусы и радианы. Следующие два сообщения выполняют преобразование:

Протокол экземпляров *Числа*

преобразование

градусы в радианы
Подразумевается что получатель представлен в градусах. Возвращает преобразование градусов в радианы.

радианы в градусы
Подразумевается что получатель представлен в радианах. Возвращает преобразование радианов в градусы.

Поэтому

```
30 градусы в радианы = 0,523599
90 градусы в радианы = 1,5708
```

Тригонометрические и логарифмические функции включены в протокол математических функций. Получатель тригонометрических функций *кос*, *син* и *тан* это угол в радианах; результат функций *арк кос* и *арк тан* это угол измеряемый в радианах.

В следующих примерах 30 градусов представлены как 0,523599 радиан, 90 градусов как 1,5708 радиан.

предложение	результат
0,523599 син.	0,5
0,523599 кос.	0,866025
0,523599 тан.	0,57735
1,5708 син.	1,0
0,57735 арк тан.	0,523551
1,0 арк син.	1,5708

Когда какой либо вид *Целого* запрашивают добавить себя к другому виду *Целого*, возвращаемый результат естественно будет тоже видом *Целого*. То же самое верно для суммы двух *Плавающих*; класс результата будет тем же что и класс операндов. Если опе-

ранды это *Малые целые* и модуль их суммы слишком велик чтобы его можно было представить в виде *Малого целого*, то результатом станет *Большое положительное целое* или *Большое отрицательное целое*. Определение подходящего класса результата при различных классах операднов немного более сложно. Принцип положенный в основу таков что теряется как можно меньше информации и что коммутативные операции возвращают тот же результат не зависимо от того какой операнд является получателем а какой аргументом. Поэтому, например, 3,1 * 4 возвратит то же что и 4 * 3,1.

Подходящее представление для результата операции над числами с различными классами определяется с помощью значения общности присвоенного каждому классу. Класс имеющий большую общность будет иметь большее число своей оценки общности. Каждый класс должен уметь преобразовывать свои экземпляры в экземпляры более общего класса с тем же значением. Мера общности используется для определения того какой из операндов нужно преобразовать. Таким образом арифметические операции подчиняются закону коммутативности без потери информации о числах. Когда различием между двумя классами чисел является только точность (где «точность» это мера информации предоставляемая числом), более точному классу присваивается большая общность в случаях когда точность не имеет значения (так *Плавающее* более общее чем *Дробъ*).

Ирархия общности для видов числе в системе Смолток с более общими классами записанными первыми:

Плавающее

Дробъ

Большое положительное целое, *Большое отрицательное целое*

Малое целое

Сообщения в протоколе *Числа* которые созданы для поддержки приведения видов содержатся в категории приведение.

Протокол экземпляров *Числа*

приведение

coerce: **aNumber**
Answer a number representing the argument, aNumber, that is the same kind of Number as the receiver. This method must be defined by all subclasses of Number.

generality
Answer the number representing the ordering of the receiver in the generality hierarchy.

retry: **aSymbol** **coercing:** **aNumber**
An arithmetic operation denoted by the symbol, aSymbol, could not be performed with the receiver and the argument, aNumber, as the operands because of the difference in representation. Coerce either the receiver or the argument, depending on which has the lower generality, and then try the arithmetic operation again. If the symbol is the equals sign, answer false if the argument is not a Number. If the generalities are the same, then retryxoercing: should not have been sent, so report an error to the user.

Поэтому если нужно выполнить 32,45 * 4, умножение *Плавающего* на *Малое целое*, то результат будет вычисляться предложением.

```
32,45 retry: #* coercing: 4
```

и аргумент 4 будет приведён к 4,0 (*Плавающее* имеет большую общность чем *Малое целое*). Затем произведение будет выполнено успешно.

Определение ирархии чисел в терминах общности численных величин работает для видов числе предоставляемых в основной системе Смолток потому что общность транзитивна для этих видов чисел. Однако, общность не предоставляет техники для использования для всех видов чисел.

Интервалы (описаны подробно в главе 10) могут быть созданы при помощи одного или двух сообщений числу. Для каждого элемента такова интервала можно выполнить блок с элементом в качестве значеня аргумента блока.

Протокол экземпляров *Числа*

интервалы

до: конец
Возвращает *Интервал* от получателя до аргумента, конец, с промежутком между элементами равным единице.

до: конец через: шаг
Возвращает *Интервал* от получателя до аргумента, конец, с промежутком между элементами равным аргументу шаг.

до: конец делать: блок
Создаёт *Интервал* от получателя до аргумента, конец, с шагом 1. Выполняет аргумент, блок, для каждого элемента интервала.

до: конец через: шаг делать: блок
Создаёт *Интервал* от получателя до аргумента, конец, с шагом шаг. Выполняет аргумент, блок, для каждого элемента интервала.

Поэтому если выполнить:

```
а ← 0.
10 до: 100 через: 10 делать: [ :каждый | а ← а + каждый. ].
```

то значение переменной а будет равно 550.

Если а это ряд #('один' 'два' 'три' 'четыре' 'пять'), то к каждому элемент этого ряда можно получить доступ при помощи номера который принадлежит интервалу от 1 до размера ряда. Следующее предложение изменяет каждый элемент так чтобы остался только первый знак.

```
1 до: а размер делать: [ :номер | а от: номер пом: ((а от: номер) от: 1). ].
```

В результате ряд станет таким: #($о $д $т $ч $п). Заметьте что подобно рядам, к элементам цепи можно получить доступ используя сообщение *от:* и *от:пом:*. Сообщения объектам подобным рядам и цепям детально рассматриваются в главах 9 и 10.

8.2 Классы *Плавающее* и *Дробь*

Классы *Плавающее* и *Дробь* это два представления не целых числовых значений. *Плавающее* представляет приближённые дей-

ствительные числа; они предоставляют точность около 6 десятичных знаков с диапазоном значений между плюс или минус 10 возведённым в степень плюс или минус 32. Вот несколько примеров:

```
8,0
13,3
0,3
2,5э6
1,27э-30
-12,987654э12
```

Дроби представляют рациональные числа которые всегда точные. Все арифметические операции с *Дробями* возвращают простые дороби.

Экземпляры *Плавающего* могут быть созданы при помощи литерала (например 3,14159) или как результат арифметических операций, один аргумент которых это другое *Плавающее*.

Экземпляры *Дроби* могут быть созданы в результате арифметической операции если один из операндов это дробь и другой не является плавающим. (Если один из аргументов это плавающее число, то результат тоже будет плавающее число т.к. общность плавающих чисел выше чем у дробей). Экземпляры *Дроби* также могут быть созданы при помощи операции деления (/) произведённой над двумя *Целыми*, если её результат не целое число. В дополнение к этим способам протокол *Дроби* поддерживает сообщение создания экземпляра *числитель: целое числ знаменатель: целое знам.*

Плавающее отвечает на сообщение *пи* возвращающее соответствующую константу. Этот класс добавляет протоколы усечения и округления для возврата дробной и целой части получателя (*дробная частъ* и *целая частъ*), и добавляет протокол преобразования для преобразования получателя в *Дробь* (*как дробъ*). Класс *Дробъ* тоже добавляет протокол преобразования для преобразования получателя в *Плавающее* (*как плавающее*).

8.3 Классы целых

Класс *Целое* добавляет протокол специфичный для целых чисел. У него есть три подкласса. Один из них то класс *Малое целое*,

который является классом важного диапазона значений с экономным использованием памяти которые часто встречаются в вычислениях и при нумерации. Представление охватывает диапазон который немного меньше чем машинное слово. Большие целые, которые представляются экземплярами *Большого положительного целого* и *Большого отрицательного целого* в зависимости от знака числа, не имеют предела величины. Цена этой общности — большее время на вычисления. Поэтому если результат арифметической операции для больших целых это целое которое можно представить при помощи малого целого, то результат будет малым целым.

В дополнение к сообщениям унаследованным от класса *Число*, класс *Целое* добавляет протокол преобразования (*как знак*, *как плавающее* и *как дробь*), также печать (*печатать в: поток основание: о*, *основание: целое основание*), и протокол перебора. Поэтому *8 основание: 2* это '2o1000'.

Для перебора, можно многократно выполнить блок число раз равное целому используя сообщение *раз повторить: блок*. Рассмотрим пример:

```
a ← 1.
10 раз повторить: [ a ← a + a. ].
```

где у блока нет аргументов. Конечное значение a это 2^{10}, или 1024.

Класс *Целое* предоставляет протоколы для факторизации и проверки делимости которые не определены для чисел в целом.

Протокол экземпляров *Целого*

факторизация и делимость

факториал
Возвращает факториал получателя. Получатель должен быть не меньше 0.

нод: **целое**
Возвращает наибольший общий делитель получателя и аргумента, целое.

нок: **целое**
Возвращает наименьшее общее кратное получателя и аргумента, целое.

Примеры:

предложение	результат
3 факториал.	6
55 нод: 30.	5
6 нок: 10.	30

В дополнение к свойствам целых, некоторые алгоритмы используют тот факт что целые можно интерпретировать как последовательность битов. Поэтому в *Целом* есть протокол для действий с битами.

Протокол экземпляров *Целого*

действия с битами

вся маска: целое
Рассматривает аргумент целое как битовую маску. Отвечает все ли быти равные 1 в маске равны 1 в получателе.

любой из маски: целое
Рассматривает аргумент целое как битовую маску. Отвечает есть ли хотя бы один бит который равный 1 и в маске и в получателе.

не маска: целое
Рассматривает аргумент целое как битовую маску. Отвечает все ли биты равные 1 в маске равны 0 в получателе.

побитовое и: целое
Возвращает целое чьи биты это логическое и соответствующих битов получателя и аргумента, целое.

побитовое или: целое
Возвращает целое чьи биты это логическое или соответствующих битов получателя и аргумента, целое.

побитовое искл или: целое
Возвращает целое чьи биты это логическое исключающее или соответствующих битов получателя и аргумента, целое.

обратить биты
Возвращает целое чьи биты это дополнение до 2 получателя.

старший бит

Возвращает номер старшего бита в двоичном представлении получателя.

сдвинуть биты: **целое**
Возвращает целое чьё значение (в представлении дополнения до двух) сдвинуто влево на количество бит указанных аргументом, целое. Отрицательный аргумент сдвигает вправо. При сдвиге влево справа подставляются нули. При сдвиге вправо происходит заполнение битом знака.

Приведём несколько примеров. Заметьте что по умолчанию основание для печати равно 10.

выражение	**результат**
2o111000111000111	29127
2o101010101010101	21845
2o101000101000101	20805
2o000111000111000	3640
29127 вся маска: 20805	истина
29127 вся маска: 21845	ложь
29127 любой из маски: 21845	истина
29127 не маска: 3640	истина
29127 побитовое и: 3640	0
29127 побитовое или: 3640	32767
32767 основание: 2	2o111111111111111
29127 побитовое или: 21845	30167
30167 основание: 2	2o111010111010111
3640 сдвинуть биты: 1	7280

8.4 Класс *Случайное число*: генератор случайных чисел

Многим программам нужен случайный выбор числа. Например, случайные числа полезны в статистике и алгоритмах шифрования. Класс *Случайное число* это генератор случайных чисел который включён в систему Смолток. Он предоставляет простой способ по-

лучения последовательности случайных чисел которые равномерно распределены на интервале между 0,0 и 1,0 (не включая границы).

Экземпляр класса *Случайное число* это источник из которого получаются случайные числа. Он инициализируется псевдо случайным образом. Когда нужно получить новое случайное число то экземпляру *Случайного числа* посылается сообщение *следующий*.

Генератор случайных чисел может быть создан при помощи следующего предложения.

```
случ ← Случайное число новый.
```

Предложение:

```
случ следующий.
```

может быть выполнено когда понадобится новое случайное число. Ответ на это сообщение это число (*Плавающее*) между 0,0 и 1,0.

Реализация метода *следующий* основана на Lehmer's linear congruential method описанном Кнутом в первом томе (Д.Э. Кнут, Искусство программирования: основные алгоритмы Том 1)

```
next
  ↑ (seed ← self nextValue) / m.
nextValue
  | lo hi aLoRHi answer |
  hi ← (seed quo: q) asFloat.
  textcolortemplo ← seed − (hi * q).
  aLoRHi ← a * lo − (r * hi).
  answer ← aLoRHi > 0,0 ifTrue: [ aLoRHi. ] ifFalse: [ aLoRHi + m. ].
  ↑ answer.
```

Также можно послать экземпляру класса *Случайное число* сообщение *следующий: целое*, чтобы получить *Упорядоченный набор* случайных чисел размером целое.

Допустим мы хотим выбрать одно из 10 целых 1, ..., 10, используя генератор случайных числе. Предложение которое нужно выполнить:

```
(случ следующий * 10) усечь + 1.
```

Здесь

выражение	**результат**
случ следующий	Случайное число между 0 и 1.
случ следующий * 10	Случайное число между 0 и 10.
(случ следующий * 10) усечь	Целое которое больше либо равно 0 и меньше либо равно 9.
(случ следующий * 10) усечь + 1	Целое которое больше либо равно 0 и меньше либо равно 10.

Глава 9

Протокол всех наборов

Оглавление

Набор это группа объектов. Эти объекты называются элементами набора. Например, *Ряд* это набор. Ряд

```
#( 'слово' 3 5 $Г #( 1 2 3 ) )
```

это набор пяти элементов. Первый элемент это *Цепъ*, второй и третий это *Малые целые*, четвёртый элемент это *Знак* и пятый элемент это *Ряд*. Первый элемент, *Цепъ*, также является набором; в данном случае это набор *Знаков*.

Наборы предоставляют простейшие структуры данный для программирования в системе Смолток. Элементы одних наборов не упорядоченны а элементы других наборов упорядочены. Набор с неупо-

Объект

- Величина
  - Знак
  - Дата
  - Время
  - Число
    - Плавающее
    - Дробь
    - Целое
      - Большое положительное целое
      - Большое отрицательное целое
      - Малое целое
  - Ключ поиска
    - Ассоциация
- Связь
  - Процесс
- **Набор**
  - Набор последовательность
    - Связанный список
      - Семафор
    - Набор ряд
      - Ряд
        - Растровое изображение
        - Ряд серий
        - Цепь
          - Символ
        - Текст
        - Ряд байтов
    - Интервал
    - Упорядоченный набор
      - Сортированный набор
  - Мешок
  - Набор отображение
  - Множество
    - Словарь
      - Тождественный словарь
- Поток
  - Позиционируемый поток
    - Поток чтения
    - Поток записи
      - Поток чтения записи
        - Поток файл
  - Случайное число
- Неопределённый объект
- Логика
  - Истина
  - Ложь
- Планировщик исполнителя
- Задержка
- Разделяемая очередь
- Поведение
  - Описание класса
    - Класс
    - Метакласс
- Точка
- Прямоугольник

рядоченными элементами *Мешок* позволяет дублирование элементов а *Множество* не позволяет дублирование. Также есть *Словари* которые связывают пары объектов. Некоторые наборы с упорядоченными элементами задают порядок при добавлении элементов (*Упорядоченный набор*, *Ряд*, *Цепь*) а другие определяют порядок на основании самих элементов (*Сортированный набор*). Например, обычные структуры данных такие как ряды и цепи представляются классами которые связывают целый номер с элементом и у них внешний порядок соответствующий порядку номеров.

Эта глава вводит протокол для всех наборов. Каждое сообщение описанное в этой главе понимается любым видом набора, если только набор не запрещает его. Описание каждого вида набора представлено в следующей главе.

Наборы поддерживают четыре категории сообщений доступа к элементам:

- сообщения для добавления нового элемента
- сообщения для удаления элементов
- сообщения для проверки присутствия элемента
- сообщения для перебора элементов

Можно добавить или удалить из набора один или несколько элементов. Можно проверить пуст ли набор или содержит ли он некоторый элемент. Также можно определить количество включений элемента в набор. Перебор позволяет получить доступ к элементам без удаления их из набора.

9.1 Добавление, удаление и проверка элементов

Основной протокол для всех наборов определяется надклассом всех классов наборов, именуемом *Набор*. Класс *Набор* это подкласс класса *Объект*. Протокол для добавления, удаления и проверки элеметов:

Протокол экземпляров *Набора*

добавление

добавить: **новый объект**
Добавляет аргумент, новый объект, как элемент получателя. Возвращает новый объект.

добавить все: **добавить все:**
Добавляет все элементы аргумента, набор, как элементы получателя. Возвращает набор.

удаление

удалить: **старый объект**
Удаляет аргумент, старый объект, из элементов получателя. Возвращает старый объект, если у получателя нет элементов равных старому объекту, то возникает ошибка.

удалить: **старый объект** **если нету:** **блок исключение**
Удаляет аргумент, старый объект, из элементов получателя. Если несколько элементов равны старому объекту, то удаляется только один. Если нет ни одного элемента равного старому объекту, то возвращается значение выполнения блока исключения. Иначе возвращается старый объект.

удалить все: **набор**
Удаляет каждый элемент аргумента, набор, из получателя. Если удаление успешно для каждого элемента аргумента, то возвращается аргумент, набор. Иначе возникает ошибка удаления.

проверки

содержит: **объект**
Отвечает равен ли аргумент, объект, хотя бы одному элементу получателя.

пустой
Отвечает не содержит ли получатель элементы.

вхождений: **объект**
Отвечает сколько элементов получателя равно аргументу, объект.

Чтобы показать использование этих сообщений введём набор *лотерея а*

(272 572 852 156)
и набор *лотерея б*
(572 621 274)

Будем подразумевать что эти два набора, представляющие числа выбранные в лотерее, являются экземплярами *Мешка*, подкласса *Набора*. Сам *Набор* это абстрактный класс в том смысле что он описывает протокол для всех наборов. *Набор* не предоставляет достаточного представления для сохранения элементов, поэтому нельзя реализовать в *Наборе* всех его сообщений. Из за незавершённости определения *Набора* нет смысла создавать экземпляры *Набора*. *Мешок* это конкретный класс в том смысле что он представляет возможность запоминания элементов и реализует сообщения которые невозможно реализовать в его надклассах.

Все наборы отвечают на сообщение *размер* которое возвращает количество элементов набора. Поэтому можно определить что

```
лотерея а размер
```

это 4 и

```
лотерея б размер
```

это 3. Выполнив предложения в следующем порядке получим:

предложение	**результат**	лотерея а *если она изменилась*
лотерея а пустой.	ложь	
лотерея а содержит: 572.	истина	
лотерея а добавить: 596.	596	Мешок (272 572 852 156 596)
лотерея а добавить все: лотерея б.	Мешок (572 621 274)	Мешок (272 274 852 156 596 572 572 621)
лотерея а вхождений: 572.	2	
лотерея а удалить: 572.	572	Мешок (272 274 852 156 596 572 621)

лотерея а размер.	7	
лотерея а удалить все: лотерея б.	Мешок (572 621 274)	Мешок (272 852 596 156)
лотерея а размер.	4	

Заметьте что сообщения *добавить:* и *удалить:* возвращают аргумент вместо самого набора, поэтому можно таким образом получить доступ к вычисленному аргументу. Сообщение *удалить:* удаляет только одно вхождение аргумента, а не все вхождения.

Блоки были введены в главе 2. Сообщение *удалить: старый объект если нету: блок исключение* — использует блок для определения поведения набора при возникновении ошибки. Аргумент *блок исключение* выполняется если объект на который ссылается переменная *старый объект* не является элементом набора. Этот блок может содержать текст обрабатывающий ошибку или просто игнорирующий её. Например, предложение

```
лотерея а удалить: 122 если нету: [].
```

ничего не делает когда определяется что 121 не является элементом *лотереи а*.

Поведение по умолчанию для сообщения *удалить:* это уведомление об ошибке при помощи посылки сообщения *ошибка: 'объект не содержится в наборе'*. (Вспомните что сообщение *ошибка:* определено в протоколе для всех объектов и поэтому понятно любому набору).

9.2 Перебор элементов

В протокол экземпляра всех наборов включены несколько сообщений для перебора которые позволяют просматривать элементы набора и передавать каждый элемент для выполнения в блок. Основное сообщение перебора это *делать: блок*. Оно получает один аргумент *блок* в качестве аргумента и выполняет блок один раз для каждого элемента набора. Например, допустим что *буквы* это набор *Знаков* и нужно узнать сколько в наборе букв а или А.

```
количество ← 0.
буквы
  делать: [
    :каждый |
    каждый как маленькая буква = $a
      истина: [ количество ← количество + 1. ]. ].
```

Эти предложения увеличивают счётчик, *количество*, на единицу для каждого элемента который является маленькой или большой буквой а. Желаемый результат это конечное значение переменной *количество*.

В протоколе всех наборов определено шесть расширений основного сообщения перебора. Описание этих сообщений перечислений указывает что создаётся «новый набор подобный получателю» для сбора результирующей информации. Эта фраза означает что новый набор это экземпляр того же класса что и класс получателя. Например, если получатель сообщения *выбрать:* это *Множество* или *Ряд*, то ответ это соответственно новое *Множество* или *Ряд*. Единственное исключение в системе Смолток для этих сообщений есть в реализации класса *Интервал*, который возвращает новый *Упорядоченный набор*, а не новый *Интервал*. Причиной данного исключения является способ создания интервала, элементы интервала создаются вместе с созданием экземпляра *Интервала*, нельзя поместить элементы в существующий интервал.

Протокол экземпляров *Набора*

перебор

делать: блок
: Выполняет аргумент, блок, для каждого элемента получателя.

выбрать: блок
: Выполняет аргумент, блок, для каждого элемента получателя. Собирает в новый набор подобный получателю только те элементы для которых блок возвратил истину. Возвращает новый набор.

отбросить: блок

Выполняет аргумент, блок, для каждого элемента получателя. Собирает в новый набор подобный получателю только те элементы для который блок возвратил ложь. Возвращает новый набор.

собрать: блок
Выполняет аргумент, блок, для каждого элемента получателя. Возвращает новый набор подобный получателю содержащий значения возвращённые блоком при каждом его выполнении.

выявить: блок
Выполняет аргумент, блок, для каждого элемента получателя. Возвращает первый элемент для которого блок возвратил истину. Если блок ни разу не возвратил истину, то выводится сообщение об ошибке.

выявить: блок **если ни одного:** блок исключение
Выполняет аргумент, блок, для каждого элемента получателя. Возвращает первый элемент для которого блок возвратил истину. Если блок ни разу не возвратил истину, то выполняется аргумент, блок исключение. Блок исключение должен быть блоком без аргументов.

ввести: это значение **в:** бинарный блок
Выполняет аргумент, бинарный блок, один раз для каждого элемента получателя. У блока есть два аргумента: второй аргумент это элемент получателя; первый аргумент это значение предыдущего выполнения блока, начальное значение этого аргумента равно аргументу, это значение. Возвращает конечное значение блока.

Каждое сообщение перебора предоставляет краткий способ выражения последовательности сообщения для проверки или сбора информации об элементах набора.

9.2.1 Выбор и отбрасывание

Можно определить сколько раз встречается буква а или А используя сообщение *выбрать:*.

```
(буквы выбрать: [ :каждый | каждый как маленькая буква = $а. ])
размер.
```

Это предложение создаёт набор содержащий только буквы a или A, и затем возвращает размер набора результата.

Так же можно определить количество букв a или A используя сообщение *отбросить:*.

```
(буквы отбросить: [ :каждый | каждый как маленькая буква ~= $a.
]) размер.
```

Здесь создаётся набор из элементов которые не являются буквой a или A, и затем возвращается размер набора результата.

Выбор между *выбрать:* и *отбросить:* должен основываться на лучшем выражении для условия выбора. Если условие выбора лучше записывается в терминах принятия элемента то проще использовать *выбрать:*, если условие выбора проще записывается в терминах отклонения то проще использовать *отбросить:*. В данном примере подходящим сообщением является *выбрать:*.

Другой пример, допустим *служащие* это набор работников, каждый из которых отвечает на сообщение оклад. Чтобы получить набор служащих с окладом по крайней мере 10000 $ нужно использовать:

```
служащие выбрать: [ :каждый | каждый оклад >= 10000. ].
```

или

```
служащие отбросить: [ :каждый | каждый оклад < 10000. ].
```

Получающиеся наборы будут одинаковыми. Выбор сообщения *выбрать:* или *отбросить:* зависит от способа которым программист хочет записать условие "по крайней мере 10000 $".

9.2.2 Собирание

Допустим нужно создать новый набор в котором собраны оклады каждого работника из набора *служащие*.

```
служащие собрать: [ :каждый | каждый оклад. ].
```

Набор результат имеет то же размер что и набор *служащие*. Каждый элемент нового набора это оклад соответствующего элемента набора *служащие*.

9.2.3 Выявление

Допустим нужно в наборе служащие найти работника с окладом болье чем 20000 $. Выражение

```
служащие выявить: [ :каждый | каждый оклад > 20000. ]
```

вернёт такого работника, если он существует. Если он не существует, то будет послано сообщение *ошибка: ' в наборе нет объекта'*. Так же как и для сообщения удаления, программист может указать поведение при неудаче сообщения *выявить:*. Следующее выражение возвращает работника чей оклад превышает 20000 $, или, если такого не существует, возвращает *пусто*.

```
служащие
  выявить: [ :каждый | каждый оклад > 20000. ]
  если ни одного: [ пусто. ]
```

9.2.4 Ввод значения

В сообщении *ввести:в:*, первый аргумент это начальное значение которое передаётся для определения результата; второй параметр это блок с двумя аргументами. Первый аргумент блока это переменная которая ссылается на результат; вторая переменная это каждый элемент набора. В примере использующем это сообщение происходит сложение окладов работников из набора *служащие*.

```
служащие
  ввести: 0
   в: [ :подсумма :след работник | подсумма + след работник
оклад. ].
```

Здесь начальное значение 0 увеличивается на значение оклада каждого работника в наборе, *служащие*. Результат это конечное значение подсуммы.

Используя сообщение *ввести:в:*, программист может задать временную переменную и избежать создания объекта в котором собираются результаты. Например, в вышеприведённом примере вычисления количества букв а или А в наборе *буквы* мы использовали счётчик, *количество*.

```
количество ← 0.
буквы
  делать: [
    :каждый |
    каждый как маленькая буква = $a
      истина: [ количество ← количество + 1. ]. ].
```

Альтернативным способом подсчёта является использование сообщения *ввести:в:*. В предложениях примера результат накапливается в переменной *количество*, *количество* вначале равно 0. Если следующая буква это a или A, то к *количеству* добавляется 1, иначе добавляется 0.

```
буквы
  ввести: 0
  в: [
    :количество :след элемент |
    количество
        + (след элемент как маленькая буква = $a истина: [ 1. ]
ложь: [ 0. ]). ].
```

9.3 Создание экземпляров

В начале этой главы, были приведены примеры в которых наборы были записаны при помощи литералов. Эти наборы были *Рядами* и *Цепями*. Например, выражение для создания ряда:

```
#('первый' 'второй' 'третий')
```

где каждый элемент это *Цепь* записаная литерально.

Для создания экземпляров данного вида набора можно использовать сообщения *новый* и *новый:*. Протокол класса для всех наборов дополнительно вводит сообщения для создания экземпляров с одним, двумя, тремя или четырьмя элементами. Эти сообщения предоставляют короткую запись для создания наборов которые нельзя записать при помощи литералов.

Протокол класса *Набор*

создание экземпляра

с: объект
Возвращает экземпляр набора содержащий объект.

с: первый объект с: второй объект
Возвращает экземпляр набора содержащий первый объект и второй объект в качестве элементов.

с: первый объект с: второй объект с: третий объект
Возвращает экземпляр набора содержащий первый объект, второй объект и третий объект в качестве элементов.

с: первый объект с: второй объект с: третий объект с: четвёртый объект
Возвращает экземпляр набора содержащий первый объект, второй объект, третий объект и четвёртый объект в качестве элементов.

Например, *Множество* это подкласс *Набора*. Чтобы создать новое *Множество* с тремя элементами буквами s, e и t, нужно выполнить предложение:

```
Множество с: $s с: $e с: $t.
```

9.4 Преобразование различных классов наборов

Полное описание и понимание допустимых преобразований между видами наборов зависит от представления всех подклассов *Набора*. Здесь только указывается что в протоколе преобразования для всех наборов есть пять сообщений для преобразования получателя в *Мешок*, *Множество*, *Упорядоченный набор* и *Сортированный набор*. Эти сообщения определены в классе *Набор* потому что можно преобразовать любой набор в любой из этих пяти видов наборов. Порядок элементов для любого неупорядоченного набора, при преобразовании в набор с упорядоченными элементами, произволен.

Протокол экземпляров *Набора*

преобразование

как мешок
Возвращает мешок с теми же элементами что и у получателя.

как множество
Возвращает множество с теми же элементами что и у получателя (однако любые повторения игнорируются).

как упорядоченный набор
Возвращает *Упорядоченный набор* с теми же элементами что и у получателя (возможно произвольное упорядочивание).

как сортированный набор
Возвращает *Сортированный набор* с теми же элементами что и у получателя, отсортированными так что каждый элемент меньше чем либо равен (<=) следующего.

как сортированный набор: блок
Возвращает *Сортированный набор* с теми же элементами что и у получателя, отсортированными в соответствии с аргументом блок.

Поэтому если *лотерея а* это *Мешок* содержащий элементы:
272 572 852 156 596 272 572 то

лотерея а как множество.

это *Множество* содержащее элементы
852 596 156 572 272
и

лотерея а как сортированный набор.

это *Сортированный набор* содержащий элементы в таком порядке:
156 272 272 572 572 596 852

Глава 10

Ирархия классов наборов

Оглавление

На рисунке 10.1 представлена схема для нахождения различных классов наборов системы. Следуя выбору на рисунке удобно определять какой вид набора использовать в реализации.

Одно из различий классов является наличие или отсутствие у набора порядка связанного с его элементами. Другое различие —

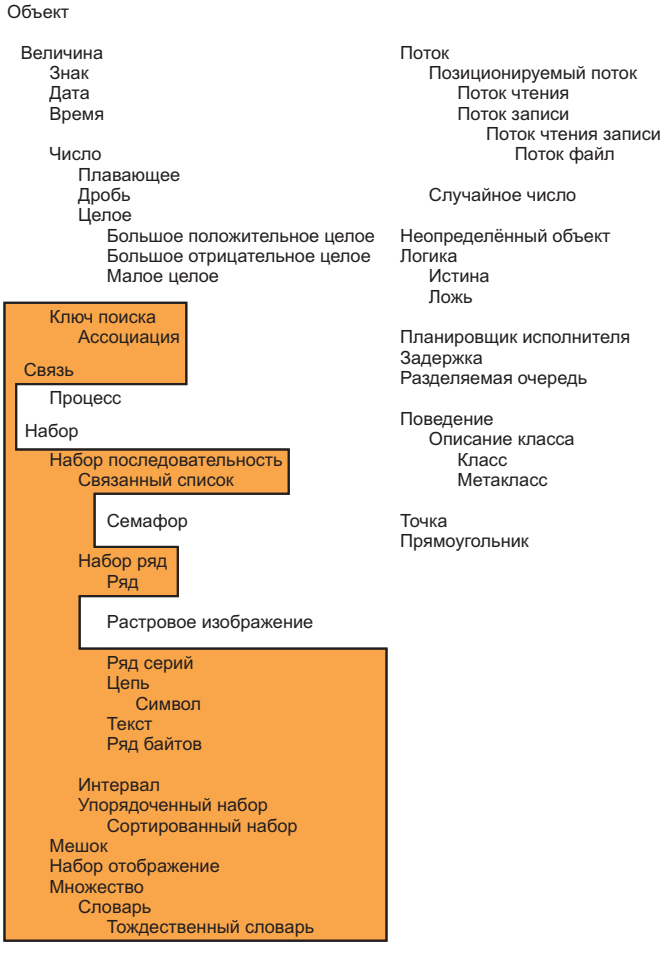

Объект
Величина
Знак
Дата
Время
Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое
Ключ поиска
Ассоциация
Связь
Процесс
Набор
Набор последовательность
Связанный список
Семафор
Набор ряд
Ряд
Растровое изображение
Ряд серий
Цепь
Символ
Текст
Ряд байтов
Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь
Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл
Случайное число
Неопределённый объект
Логика
Истина
Ложь
Планировщик исполнителя
Задержка
Разделяемая очередь
Поведение
Описание класса
Класс
Метакласс
Точка
Прямоугольник

Рис. 10.1

можно ли получить доступ к элементам через внешний ключ или имя. Тип ключа определяет другое различие между видами наборов. Некоторые ключи это целые числа, явно сопоставленные с порядком элементов; другие ключи неявно связывают объекты которые служат в качестве ключа поиска.

Один из неупорядоченных наборов с внешними ключами это *Словарь*. Его ключами обычно являются экземпляры *Цепи* или *Ключа поиска*; для проверки ключей на совпадение используется равенство (=). Словарь имеет подкласс, *Тождественный словарь* чьи внешние ключи обычно являются *Символами*. У него проверка ключей на совпадение это эквивалентность (==). Элементы *Мешка* или *Множества* не упорядочены и не доступны через внешние ключи. Дублирование позволяется в *Мешке*, но не во *Множестве*.

Все упорядоченные наборы это виды *Набора последовательности*. Элементы всех *Наборов последовательностей* доступны через ключ целое число. Четыре подкласса *Набора последовательности* предоставляют различные способы создания порядка элементов. Дополнительное различие между классами *Набора последовательности* это возможность хранить элементы которые являются экземплярами любого класса или экземплярами некоторого вида объектов.

Порядок элементов определяется внешним образом для *Набора последовательности*, *Связанного списка* и *Набора ряда*. Для *Набора последовательности* и *Связанного списка* последовательность действий программы по добавлению и удалению элементов определяет порядок элементов. Элементом *Набора последовательности* может быть любой объект, в то время как элементом *Связанного списка* может быть только вид *Связи*. Различные *Наборы ряды* системы включают *Ряд*, *Цепь* и *Ряд байтов*. Элементами *Ряда* или *Ряда серий* может быть объект любого вида, элементами *Цепи* или *Текста* должны быть *Знаки*, и элементами *Ряда байтов* должны быть *Малые целые* между 0 и 255.

Порядок элементов определяется самими элементами для *Интервалов* и *Сортированных наборов*. Для *Интервалов*, порядок это арифметическая прогрессия которая задаётся при создании экземпляра. Для *Сортированного набора*, порядок определяется по условию сортировки заданному блоком известным набору. Элемен-

тами *Интервала* должны быть *Числа*; элементами *Сортированного набора* могут быть любые виды объектов.

В дополнение к уже упомянутым классам наборов существует и *Набор отображение*, это набор который предоставляет косвенный доступ к набору чьи элементы доступны через внешний ключ. Это отображение одного множество внешних ключей на другое задаваемое во время создания *Набора отображения*.

Дальше в этой главе рассматривается каждый из подклассов наборов, описывается добавление к протоколу сообщений и приводятся простые примеры.

10.1 Класс *Мешок*

Мешок это простейший вид набора. Он представляет собой набор чьи элементы не упорядочены и не имеют внешних ключей. Это подкласс *Набора*. Т.к. его экземпляры не имеют внешних ключей, то он не отвечает на сообщения *от:* и *от:пом:*. Сообщение размер возвращает общее количество элементов в наборе.

Мешок это просто группа элементов который ведёт себя соответственно протоколу для всех наборов. Общее описание наборов не ограничивает количество повторений элемента в наборе. Класс *Мешок* подчёркивает эту общность определяя дополнительное сообщение для добавления элементов.

Протокол экземпляров *Мешка*

добавление

добавить: новый объект с вхождениями: целое
Включает аргумент, новый объект, как элемент получателя, целое число раз. Возвращает аргумент, новый объект.

Рассмотрим класс пример *Продукт* который представляет бакалейный товар и его цену. Новый продукт может быть создн при помощи сообщения *с: имя за: цена*, и цена экземпляра доступна при помощи сообщения *цена*. Заполнение корзины можно представить так:

```
пакет ← Мешок новый.
пакет добавить: (Продукт с: #steak за: 5.80).
пакет добавить: (Продукт с: #potatoes за: 0.50) с вхождениями: 6.
пакет добавить: (Продукт с: #carrots за: 0.10) с вхождениями: 4.
пакет добавить: (Продукт с: #milk за: 2.20).
```

Затем можно определить плату за покупки при помощи предложений:

```
количество ← 0.
пакет
    делать: [ :каждый продукт | количество ← количество +
каждый продукт цена. ].
```

или

```
пакет
  ввести: 0
  в: [ :количество :каждый продукт | количество + каждый
продукт цена. ].
```

получится 11.40 $. Заметьте что сообщения *добавить:*, *делать:* и *ввести:в:* наследуются *Мешком* от своего надкласса, *Набора*.

Мешок является неупорядоченным, т.е. не смотря на то что поддерживаются сообщения перебора не известно в каком порядке будут перебираться элементы.

10.2 Класс *Множество*

Класс *Множество* представляет набор чьи элементы неупорядочены и не имеют внешних ключей. Его экземпляры не отвечают на сообщение *от:* и *от:пом:*. *Множество* подобно *Мешку* за исключением того что его элементы не могут дублироваться. Сообщение добавления добавляет элемент только если он ещё не в наборе. Класс *Множество* это подкласс класса *Набор*.

10.3 Классы *Словарь* и *Тождественный словарь*

Класс *Словарь* представляет набор связей между ключами и значениями. Элементы словаря это экземпляры класса *Ассоциация*, простейшей структуры данный для хранения пары ключ-значение.

Альтернативным способом представлять себе *Словарь* можно как набор чьи элементы не упорядочены но явно доступны при помощи ключей или имён. С этой перспективы, элементы *Словаря* это произвольные объекты (значения) с внешними ключами. Альтернативный способ представления о *Словаре* отражён в протоколе сообщений класса. Сообщения наследуемые от класса *Набор* — *включает:*, *делать:* и другие сообщения для перечисления применяются к значениям *Словаря*. То есть эти сообщения используют значение каждой ассоциации в *Словаре*, а не ключ или саму ассоциацию.

Сообщения унаследованные от класса *Объект* — *от:* и *от:пом:* — применяются к ключам *Словаря*. Принцип сообщений *от:* и *от:пом:* расширяется для ассоциаций и значений при помощи дополнительный сообщений *ассоциация от:* и *ключ от значения:*. Для предоставления дополнительного контроля при поиске элемента в *Словаре* есть сообщение *от:если нету:*, используя его программист может задать действие в случае если элемент с ключём равным первому аргументу не найден. Наследуемое сообщение *от:* сообщает об ошибке если ключ не найден.

Протокол экземпляров *Словаря*

доступ

от: ключ если нету: блок
Возвращает значение именованное аргументом, ключ. Если ключ не находится, то возвращается результат выполнения блока.

ассоциация от: ключ
Возвращает ассоциацию именуемую аргументом, ключ. Если ключ не находится, то сообщается об ошибке.

ассоциация от: ключ если нету: блок
Возвращает ассоциацию именуемую аргументом, ключ. Если ключ не находится, то возвращается результат выполнения блока.

ключ от значения: значение
Возвращает имя аргумента, значение. Если нет такого значения, возвращает пусто. Т.к. значение не должно быть уникальным, то возвращается первое найденное имя.

ключ от значения: значение если нету: блок исключение
Возвращает ключ для аргумента, значение. Если такого значения нет, то возвращается значение выполнения блока исключения.

ключи
Возвращает *Множество* содержащие ключи получателя.

значения
Возвращает *Мешок* содержащий значения получателя (включая все повторения).

В качестве примера использования *Словаря* рассмотрим словарь *Символов* антонимов.

```
антонимы ← Словарь новый.
антонимы от: #горячий пом: #холодный.
антонимы от: #толкать пом: #тянуть.
антонимы от: #стой пом: #иди.
антонимы от: #приходи пом: #иди.
```

Также можно добавить элемент при помощи сообщения *добавить:* задав *Ассоциацию* в качестве аргумента.

```
антонимы ← Словарь новый.
антонимы добавить: (Ассоциация ключ: #перед значение: #зад).
антонимы добавить: (Ассоциация ключ: #верх значение: #низ).
```

Словарь *антонимы* сейчас содержит:

ключ	значение
горячий	холодный
толкать	тянуть
стой	иди
приходи	иди
перед	зад
вехр	низ

Воспользовавшись протоколом проверки унаследованным от класса *Набор* проверим значения словаря. Заметьте что *включает:* проверяет присутствие значения, а не ключа.

предложение	результат
антонимы размер.	6
антонимы содержит: #холодный.	истина
антонимы содержит: #горячий.	ложь
антонимы вхождений: #иди.	2
антонимы от: #стой пом: #старт.	старт

Четвёртый пример показывает что не смотря на то что ключ может встречаться только один раз значение может быть связано с любым количеством ключей. Последний пример пересвязывает ключ #стой с новым значением #старт. Классом *Словарь* предоставляются дополнительные сообщения для проверки ассоциаций и ключей.

Протокол экземпляров *Словаря*

проверка словаря

содержит ассоциацию: **ассоциация**

Отвечает содержит ли получатель элемент (ассоциацию между ключём и значением) который равен аргументу, ассоциация.

содержит ключ: **ключ**

Отвечает есть ли у получателя ключ равный аргументу, ключ.

Поэтому можно проверить:

предложение	результат
антонимы содержит ассоциацию: (Ассоциация ключ: #приходи значение: #иди).	истина
антонимы содержит ключ: #горячий.	истина

Протокол удаления определённый в классе *Набор* также расширен для предоставления доступа к ассоциациям и ключам. Однако, сообщение *удалить:* не подходит *Словарю*, чтобы удалить элемент нужно указывать ключ.

Протокол экземпляров *Словаря*

удаление из словаря

удалить ключ: **ключ**

Удаляет ключ (и его ассоциацию) из получателя. Если в получателе нету ключа, сообщается об ошибке. Иначе возвращается значение связанное с ключём.

удалить ключ: **ключ** **если нету:** **блок**

Удаляет ключ (и его ассоциацию) из получателя. Если в получателе нету ключа, то возвращается результат выполнения блока. Иначе возвращается значение именуемое ключём.

Например:

предложение	результат
антонимы удалить ключ: #горячий.	Ассоциация чей ключ это #горячий и чьё значение это #холодный. Ассоциация удаляется из антонимов.

антонимы удалить ключ: #холодный если нету: [антонимы от: #холодный пом: #горячий.].	

Результатом последнего примера является включение ассоциации #холодный с #горячий в антонимы.

Сообщение *делать:* выполняет свой аргумент, блок, для каждого значения *Словаря*. Протокол перебора набора, унаследованный от класса *Набор*, снова расширяется для предоставления сообщений перебирающих ассоциации и ключи. Сообщения поддерживающие использование *отбросить:* и *ввести:в:* не предоставляются.

Протокол экземпляров *Словаря*

перебор словаря

ассоциации делать: блок
Выполняет блок для каждой ассоциации получателя между ключём и значением.

ключи делать: блок
Выполняет блок для каждого ключа получателя.

Таким образом есть три пути для перебора элементов в *Словаре*. Допустим новые слова это *Множество* слов которые дети ещё не выучили. Любое слово находящиеся в антонимах используется детьми и может быть удалено из новых слов. Выполние двух следующих предложений удаляет эти слова (первое удаляет значение, второе ключ).

```
антонимы делать: [ :слово | новые слова удалить: слово если нету: [ ]. ].
антонимы
    ключи делать: [ :слово | новые слова удалить: слово если нету: [ ]. ].
```

Заметьте что если слова из антонимов нет в новых словах, то

ничего не происходит (нет сообщения об ошибке). Также можно использовать одно предложение перебирающее *Ассоциации*.

```
антонимы
  ассоциации делать: [
    :каждая |
    новые слова удалить: каждая ключ если нету: [ ].
    новые слова удалить: каждая значение если нету: [ ]. ].
```

Сообщения доступа ключи и значения могут быть использованы для получения наборов слов из словаря *антонимы*. Предполагая что все предыдущие примеры выполнены, получим:

```
антонимы ключи
```

возвратит *Набор* чьими элементами будут:
толкать приходи перед стой холодный
и

```
антонимы значения
```

возвратит *Мешок* чьи элементы это:
тяни иди зад старт горячий

10.4 Класс *Набор последовательность*

Класс *Набор последовательность* представляет набор чьи элементы упорядоченны и поименованы снаружи целыми указателями. *Набор последовательность* это подкласс *Набора* и он предоставляет протокол для доступа, копирования и перечисления элементов набора когда известно что есть порядок связанный с элементами. Т.к. элементы упорядочены, то есть определённый первый и последний элементы набора. Можно спросить номер некоторого элемента (*номер для:*) и номер начала последовательности набора элементов (*номер поднабора:начиная с:*). Все наборы наследуют сообщения для доступа к нумерованным переменным от класса *Объект*. Как описано в главе 6, это *от:*, *от:пом:* и *размер*. Дополнительно *Набор последовательность* поддерживает помещение объектов на все позиции именуемые элементами *Набора* (*от всех:пом все:*), и помещение объекта во все позиции последовательности (*от всех*

пом:). Последовательность элементов набора можно заменить элементами другой последовательности (*заменить от:до:на:* и *заменить от:до:на:начиная с:*)

Протокол экземпляров *Набора последовательности*

доступ

от всех: набор пом: объект
Связывает каждый элемент аргумента, набор (целое или другой внешний ключ), со вторым аргументом, объект.

от всех пом: объект
Помещает аргумент, объект, в качестве каждого элемента получателя.

первый
Возвращает первый элемент получателя. Сообщает об ошибке если у получателя нету элементов.

последний
Возвращает последний элемент получателя. Сообщает об ошибке если у получателя нету элементов.

номер для: элемент
Возвращает первый номер аргумента, элемент, в получателе. Если получатель не содержит элемента, то возвращается 0.

номер для: элемент если нету: блок исключение
Возвращает первый номер аргумента, элемент, в получателе. Если получатель не содержит элемента, то возвращается результат выполнения аргумента блок исключение.

номер поднабора: поднабор начиная с: номер
Если элементы аргумента, поднабор, встречаются в данном порядке в получателе, то возвращается номер первого элемента первого вхождения. Если совпадений нету, то возвращается 0.

номер поднабора: поднабор начиная с: номер если нету: блок исключение

Возвращет номер первого элемента получателя такого что этот элемент равен первому элементу аргумента, поднабор, и последующие элементы равны оставшимся элементам поднабора. Поиск в получателе начинается с элемента с номером равным аргументу, номер. Если совпадений не находится, то возвращается результат выполнения аргумента блок исключение.

заменить от: **начало** **до:** **конец** **на:** **набор замена**
Связывает каждый номер между началом и концом с элементами аргумента, набор замена. Возвращает получателя. Количество элементов в наборе замене должно совпадать со значением конец − начало + 1.

заменить от: **начало** **до:** **конец<** **на:** **набор замена** **начиная с:** **начало замены**
Связывает каждый номер между началом и концом с элементами аргумента, набор замена, начиная с элемента набора замены чей номер это начало замены. Возвращает получателя. Проверка границ не производится, за исключением если получатель такой же как набор замена но начало замены не 1, тогда произойдёт сообщается об ошибке выхода за границы.

Примеры использования методов доступа, используется экземляры "Цепи".

предложение	**результат**
'аааааааааа' размер.	10
'аааааааааа' от всех: (2 до: 10 через: 2) пом: $б.	'абабабабаб'
'аааааааааа' от всех пом: $б.	'бббббббббб'
'Эта цепь' первый.	$Э
'Эта цепь' последний.	$ь
'АБВГДЕЁЖЗИКЛМНОП' номер для: $Е.	6
'АБВГДЕЁЖЗИКЛМНОП' номер для: $М если нету: [0.].	13
'АБВГДЕЁЖЗИКЛМНОП' номер для: $Я если нету: [0.].	0

'The cow jumped' номер поднабора: 'cow' начиная с: 1.	5
'The cow jumped' заменить от: 5 до: 7 на: 'dog'.	'The dog jumped'
'The cow jumped' заменить от: 5 до: 7 на: 'the spoon ran' начиная с: 5.	'The spo jumped'

Любой из этих примеров должен работать подобным образом с любым экземпляром подкласса *Набора последовательности*, например, с *Рядом*. Для Ряда, #(#The #brown #jug), замена brown на black производится предложением:

#(#The #brown #jug) заменить с: 2 до: 2 на: #(#black)

Заметьте что последний аргумент должен быть *Рядом*. Так же заметьте что сообщение замена не изменяет размер исходного набора (получателя), хотя оно изменяет набор. Может понадобиться сохранить исходный набор создав его копию. Протокол копировани *Набора последовательности* поддерживает копирование последовательности элементов набора, копирование всего набора с заменой его части, копирование всего набора с удалением элементов или копирование всего набора с добавлением одного или нескольких элементов.

Протокол экземпляров *Набора последовательности*

копирование

, набор последовательность
Это операция соединения. Возвращает копию получателя с добавленными элементами аргумента, набор последовательность.

копия от: начало до: конец
Возвращает копию подмножества получателя, начиная с элемента с номером начало до элемента с номером конец.

копия с заменой всех: старый поднабор на: новый поднабор
Возвращает копию получателя в которой все вхождения старого поднабора заменены на новый поднабор.

копия с заменой от: начало до: конец на: набор замена
Возвращает копию получателя удовлетворяющую следующим условиям: если конец меньше чем начало, то это вставка; конец должен быть точно равен началу − 1. начало = 1 означает что это вставка перед первым элементом, начало = размер + 1 означает добавление после последнего элемента. Иначе это замена; начало и конец должны лежат в пределах получателя.

копия с: новый элемент
Возвращает копию получателя которая больше на 1 элемент, новый элемент, добавленный в конец.

копия без: старый элемент
Возвращает копию получателя в которой нет вхождений старого элемента.

Используя сообщения копирования и замены можно сделать простой редактор текста. Система Смолток содержит класс *Цепь* и также класс *Текст*, последний класс предоставляет поддержку для связывания знаков в цепи со шрифтом или выделением чтобы смешивать шрифты, выделение жирным, наклонным и подчёркиванием. Протокол сообщений для *Текста* такой же как и для *Набора последовательности* с дополнениями позволяющими задавать выделение. Для иллюстрации мы используем экземпляр класса *Цепь*, но подразумеваем подобную программу для редактирования сообщений использующую экземпляры класса *Текст*. Предположим что *строка* изначально это пустая цепь.

строка ← Цепь новый: 0.

Тогда

предложение	результат
строка ← строка копия с заменой от: 1 до: 0 на: 'this is the first line tril'.	'this is the first line tril'
строка ← строка копия с заменой всех: 'tril' на: 'trial'.	'this is the first line trial'
строка ← строка копия с заменой от: строка размер + 1 до: строка размер на: 'and so on'.	'this is the first line trialand so on'
строка номер поднабора: 'trial' начиная с: 1.	24
строка ← строка копия с заменой от: 29 до: 28 на: ' '.	'this is the first line trial and so on'

Два последних сообщения протокола копирования, описанного выше, полезны для получения копий "Ряда" с или без элемента. Например:

предложение	результат
#(#один #два #три) копия с: #четыре.	(один два три четыре)
#(#один #два #три) копия без: #два.	(один три)

Из за того что элементы *Набора последовательности* упорядочены, перебор тоже упорядочен, начинается с первого элемента и идёт к следующему элементу до последнего. Также возможен перебор в обратном порядке, используя сообщение *реверсивно делать: блок*. Перебор двух *Наборов последовательностей* можно осуществить вместе так чтобы пары элементов, один из каждого набора, использовались в выполнении блока.

Протокол экземпляров *Набора последовательности*

перебор

найти первый: блок
Выполняет блок для каждого элемента получателя. Возвращает номер первого элемента для которого блок вернул истину.

найти последний: блок
Выполняет блок для каждого элемента получателя. Возвращает номер последнего элемента для которого аргумент, блок, вернул истину.

реверсивно делать: блок
Выполняет блок для каждого элемента получателя, начиная от последнего элемента и последовательно до первого элемента. Для Набора последовательности это обратный перебор по сравнению с делать:. блок это блок с одним аргументом.

с: набор последовательность **делать:** блок
Выполняет блок для каждого элемента получателя вместе с соответствующим элементом из набора последовательности. набор последовательность должен быть того же размера что и получатель, и блок должен быть двухаргументным блоком.

Следующие предложения создают *Словарь*, *антонимы*, который был использован в предыдущих примерах.

```
антонимы ← Словарь новый.
#( #приходи #холодный #перед #горячий #толкать #стой )
  с: #( #иди #горячий #зад #холодный #тянуть #старт )
  делать: [ :ключ :значение | антонимы от: ключ пом: значение.
].
```

Сейчас *Словарь* имеет шесть ассоциаций в качестве элементов.

Любой *Набор последовательность* может быть преобразован в *Ряд* или *Набор отображение*. Сообщения делающие это: *как ряд* и *mappedBy: aSequenceableCollection*.

10.5 Подклассы *Набора последовательности*

Подклассы "Набора последовательности" это "Упорядоченный набор", "Связанный список", "Интервал" и "MappedCollection". "Набор ряд" это подкласс представляющий набор элементов с фиксированным диапазоном целых в качестве внешних ключей. Подклассы "Набора рядя" это, например, "Ряд" и "Цепь".

10.5.1 Класс *Упорядоченный набор*

Упорядоченыый набор упорядочивается при помощи последовательности в которой объекты добавляются и удаляются из него. Элементы доступны через внешние ключи являющиеся номерами. Протоколы доступа, добавления и удаления расширены для ссылания на первый и последний элементы, и на элементы предшествующие или следующие за другим элементом.

Упорядоченный набор может работать как стэк или очередь. Стэк это последовательный список для которого все добавления и удаления производятся с одной стороны списка (называемой или тыл или фронт). Стэк часто описывают выражением: последним вошёл первым вышел.

обычный словарь	**сообщения *Упорядоченного набора***
поместить новый объект	**добавить последним: новый объект**
извлечь	**удалить последний**
вершина	**последний**
пустой	**пустой**

Очередь это последовательный список для которого все добавления происходят с одной стороны списка (тыл), но все удаления производятся с другой стороны (фронт). Очередь часто описывается выражением: первым пришёл последним ушёл.

обычный словарь	**сообщения *Упорядоченного набора***
добавить новый объект	**добавить последним: новый объект**
удалить	**удалить первый**
фронт	**первый**
пустой	**пустой**

Для *Упорядоченного набора* сообщение *добавить:* означает «добавить элемент в набор последним» и *удалить:* означает «удалить элемент равный аргументу». Протокол сообщений для *Упорядоченного набора*, в дополнение к унаследованному от классов *Набор* и *Набор последовательность*, включает:

Протокол экземпляров *Упорядоченного набора*

доступ

после: старый объект
Возвращает элемент идущий в получателе после старого объекта. Если получатель не содержит старого объекта или если получатель не содержит после старого объекта элементов, то сообщается об ошибке.

перед: старый объект
Возвращает элемент идущий в получателе перед старым объектом. Если получатель не содержит старый объект или если в получателе перед старым объектом нет элементов, то сообщается об ошибке.

добавление

добавить: новый объект после: старый объект
Добавляет аргумент, новый объект, как элемент получателя. Помещает его в последовательности сразу после старого объекта. Возвращает новый объект. Если старый объект не находится, то сообщается об ошибке.

добавить: новый объект перед: старый объект

Добавляет аргумент, новый объект, как элемент получателя. Помещает его в последовательности сразу перед старым объектом. Возвращает новый объект. Если старый объект не находится, то сообщается об ошибке.

добавить первыми все: олдж
Добавляет каждый элемент аргумента, упорядоченный набор, в начало получателя. Возвращает аргумент.

добавить последними все: упорядоченный набор
Добавляет каждый элемент аргумента, упорядоченный набор, в конец получателя. Возвращает аргумент.

добавить первым: новый объект
Добавляет аргумент, новый объект, в начало получателя. Возвращает аргумент.

добавить последним: новый объект
Добавляет аргумент, новый объект, в конец получателя. Возвращает аргумент.

удаление

удалить первый
Удаляет первый элемент получателя и возвращает его. Если получатель пуст, то сообщается об ошибке.

удалить последний
Удаляет последний элемент получателя и возвращает его. Если получатель пуст, то сообщается об ошибке.

10.5.2 Класс *Сортированный набор*

Класс *Сортированный набор* это подкласс *Упорядоченного набора*. Элементы *Сортированного набора* упорядочиваются функцией двух аргументов. Функция представляетя блоком называемым *сортирующий блок*. Добавление элемента возможно только сообщением *добавитъ:*; такие сообщения как *добавитъ последним:* которые позволяют программисту определять порядок вставки не допускаются для *Сортированного набора*.

Экземпляр класса *Сортированный набор* можно создать послав *Сортированному набору* сообщения *сортирующий блок:*. Аргументом для этого сообщения служит двухаргументный блок, например:

```
Сортированный набор сортирующий блок: [ :а :б | а <= б. ].
```

Данный блок это функция сортировки по умолчанию для случаев когда экземпляр создаётся при помощи сообщения *новый*. Таким образом четыре примера создания *Сортированного набора* это:

```
Сортированный набор новый.
Сортированный набор сортирующий блок: [ :а :б | а > б. ].
любой набор как сортированный набор.
любой набор как сортированный набор: [ :а :б | а > б. ].
```

Можно задать блок и сбросить блок используя два дополнительных сообщения к экземпляру *Сортированного набора*. Когда блок изменяется элементы набора, конечно же, пересортировываются. Заметьте что то же самое сообщение посылается классу (*сортирующий блок:*) для создания экземпляра с заданным условием сортировки, и экземпляру для изменения его условия.

Протокол *Сортированного набора*

методы класса

создание экземпляра

сортирующий блок: олдж
Возвращает экземпляр Сортированного набор такой что его элементы будут отсортированы в соответствии с условием заданным аргументом, блок.

методы экземпляра

доступ

сортирующий блок
Возвращает блок который является условием сортировки элементов получателя.

сортирующий блок: блок
Делает аргумент, блок, условием сортировки элементов получателя.

Допустим нам нужен алфавитный список имён детей из класса.

```
дети ← Сортированный набор новый.
```

Начальное условие сортировки это блок по умолчанию [:а :б | а <= б.]. Элементами набора могут быть *Цепи* или *Символы* потому что, как будет сказано позднее, эти виды объектов отвечают на сообщения сравнения <, >, <= и >=.

предложение	результат
дети добавить: #Joe.	Joe
дети добавить: #Bill.	Bill
дети добавить: #Alice.	Alice
дети.	Сортированный набор (Alice Bill Joe)
дети добавить: #Sam.	Sam
дети сортирующий блок: [:а :б | а < б.].	Сортированный набор ((Sam Joe Bill Alice)

дети добавить: #Henrietta.	Henrietta
дети.	Сортированный набор (Sam Joe Henrietta Bill Alice)

Шестое сообщение примера изменяет порядок в котором элементы хранятся в наборе дети.

10.5.3 Класс *Связанный список*

Связанный список это другой подкласс *Набора последовательности* чьи элементы явно упорядочиваются порядком в котором они добавляются или удаляются из набора. Подобно *Упорядоченному набору*, на элементы *Связанного списка* можно ссылаться при помощи номеров. В отличии от *Упорядоченного набора*, чьими элементами могут быть любые объекты, элементы *Связанного списка* однородны; каждый элемент должен быть экземпляром класса *Связъ* или его подкласса.

Связъ это запись ссылки на другую *Связъ*. Её протокол сообщений содержит три сообщения. То же самое сообщение (*следующая ссылка:*) используется для создания экземпляра *Связи* с данной ссылкой, и для изменения ссылки экземпляра.

Протокол класса *Связанный список*

методы класса

создание экземпляра

следующая связь: **связь**
Создаёт экземпляр *Связи* который ссылается на аргумент, связь.

методы экземпляра

доступ

следующая связь
Возвращает ссылку получателя.

следующая связь: **связь**

Присваивает ссылке получателя аргумент, связь.

Т.к. класс *Связь* не предоставляет способа записи действительного элемента набора, то он рассматривается как абстрактный класс. Поэтому, экземпляры этого класса не создаются. Вместо этого определяются подклассы которые предоставляют механизм для запоминания одного или нескольких элементов, и создаются экземпляры подклассов.

Т.к. *Связанный список* это подкласс *Набора последовательности*, то его экземпляры могут отвечать на сообщения доступа, добавления, удаления и перебора определённые для всех наборов. Дополнение протокола для *Связанного списка* состоит из:

Протокол экземпляра *Связанного списка*

добавление

добавить первым: **связь**

Добавляет связь в начало списка получателя. Возвращает связь.

добавить последним: **связь**

Добавляет связь в конец списка получателя. Возвращает связь.

удаление

удалить первый

Удаляет первый элемент получателя и возвращает его. Если получатель пуст, то сообщается об ошибке.

удалить последний

Удаляет последний элемент получателя и возвращает его. Если получатель пуст, то сообщается об ошибке.

Пример подкласса *Связи*, из системы Смолток, это класс *Процесс*. Класс *Семафор* это подкласс *Связанного списка*. Эти два класса обсуждаются в главе 15, которая рассказывает о различных независимых процессах в системе.

Дальше идёт пример использования *Связанного списка*. *Связь* не предоставляет информации кроме ссылки на другую связь. Поэтому, для примера, предположим что есть подкласс *Связи* с именем *Запись*. Запись добавляет возможность хранить объект. Сообщения создания экземпляра *Записи* это *для: объект*, и её сообщение доступа это *элемент*.

имя класса Запись
надкласс Связь
имена переменных экземпляра элемент

методы класса

создание экземпляра

для: объект
↑сам новый задать элемент: объект.

методы экземпляра

доступ

элемент
↑элемент.

печать'

печатать в: поток
поток пом следующими все: 'Entry for: ', элемент цепь для печати.

собственные

присвоить элементу: объект
элемент ← объект.

Затем классы *Связанный список* и *Запись* можно использовать так:

предложение	результат
список ← Связанный список новый.	Связанный список ()

список добавить: (Запись для: 2).	Запись для: 2
список добавить: (Запись для: 4).	Запись для: 4
список добавить последним: (Запись для: 5).	Запись для: 5
список добавить первым: (Запись для: 1).	Запись для: 1
список.	Связанный список (Запись для: 1 Запись для: 2 Запись для: 4 Запись для: 5)
список пустой.	ложь
список размер.	4
список ввести: 0 в: [:значение :каждый | каждый элемент + значение.].	12
список последний.	Запсь для: 5
список первый.	Запись для: 1
список удалить: (Запись для: 4).	Запись для: 4
список удалить первый.	Запись для: 1
список удалить последний.	Запись для: 5
список первый == список последний.	истина

10.5.4 Класс *Интервал*

Другой вид *Набора последовательности* это набор чисел представляющий математическую прогрессию. Например, набор может содержать все целые в интервале от 1 до 100; или он может содержать все чётные целые в интервале от 1 до 100. Или набор может содержать последовательность чисел в которой каждое следующее число последовательности вычисляется из предыдущего умножением его на 2. Последовательность может начинаться с 1 и заканчиваться числом которое меньше или равно 100. Это будет последовательность 1, 2, 4, 8, 16, 32, 64.

Математическая прогрессия описывается первым числом, пределом (максимум или минимум) для последнего вычисляемого числа, и методом вычисления следующего числа. Предел может быть по-

ложительной или отрицательной бесконечностью. В случае арифметической прогрессии метод вычисления это просто добавление приращения. Например, это может быть последовательность чисел в которой каждое следующее число вычисляется из предыдущего добавлением -20. Последовательность может начинаться со 100 и заканчиваться числом большим либо равным 1. Это должна быть последовательность 100, 80, 60, 40, 20.

В системе Смолток класс последовательности называемый *Интервал* содержит конечные арифметические прогрессии. В дополнение к сообщениям наследуемым от надкласса *Набор последовательность* и *Набора*, класс *Интервал* поддерживает сообщения для доступа к значениям которые описывает экземпляр. Новые элементы не могут быть добавлены или удалены из *Интервала*.

Протокол класса *Интервал* содержит следующие сообщения для создания экземпляров.

Протокол класса *Интервал*

создание экземпляра

от: **начальное целое** **до:** **конечное целое**
Возвращает экземпляр класса *Интервал*, начинающийся с числа начальное целое, заканчивающийся числом конечное целое и использующий для вычисления следующего элемента приращение 1.

от: **начальное целое** **до:** **конечное целое** **через:** **целое шаг**
Возвращает экземпляр *Интервала*, начинающийся с числа начальное целое, заканчивающийся числом конечное целое и использующий приращение целое шаг для вычисления следующего элемента.

Интервалу могут быть посланы все сообщения понимаемые *Набором последовательностью*. В дополнение к этому, протокол экземпляра *Интервала* предоставляет сообщения для доступа к шагу арифметической прогрессии (*шаг*).

Класс *Число* предоставляет два сообщения которые являются сокращениями для создания новых *Интервалов*: *до: конец* и *до: конец через: шаг*. Поэтому чтобы создать *Интервал* целых от 1 до

10, нужно выполнить:

```
Интервал от: 1 до: 10.
```

или

```
1 до: 10.
```

Чтобы создать *Интервал* начинающийся со 100 и заканчивающийся 1, добавляющий -20, нужно выполнить:

```
Интервал от: 100 до: 1 через: -20.
```

или

```
100 до: 1 через: -20.
```

Это последоватеьность 100, 80, 60, 40, 20. *Интервал* может содержать не только *Целые*; чтобы создать *Интервал* между 10 и 40, с шагом 0,2, выполните:

```
Интервал от: 10 до: 40 через: 0,2.
```

или

```
10 до: 40 через: 0,2.
```

Это последовательность 10, 10,2, 10,4, 10,6, 10,8, 11,0, ... и т.д.

Заметьте что можно создать более общий вид прогрессии заменив численное значение шага блоком. Когда новый элемент будет вычислен, он будет послан в качестве аргумента сообщения *значение:* блоку.

Сообщение *делать:* посылаемое *Интервалу* делает то же что обычный цикл for в языках программирования. Выражения на Алголе

```
for i := 10 step 6 until 100 do begin < statements > end
```

представляется так:

```
(10 до: 100 через: 6) делать: [ :н | <предложения>. ].
```

Числа отвечают на сообщения *до:через:делать:* хотя предложения можно писать и как в данном примере. Поэтому итерация может быть записана без скобок:

```
10 до: 100 через: 6 делать: [ :н | <предложения>. ].
```

Чтобы увеличить на 1 каждый шестой элемент *Упорядоченного набора* выполните:

```
6
  до: числа размер
  через: 6
  делать: [ :номер | числа от: номер пом: (числа от: номер) + 1. ].
```

Созданный *Интервал* состоит из 6, 12, 18, . . . , до номера последнего элемента чисел. Если размер набора меньше чет 6 (подразумеваемый первый номер), то ничего не происходит. Иначе элемента в позициях 6, 12, 18, и т.д., до последнего возможного элемента заменяются на новые.

10.6 Класс *Набор ряд*

Как было сказано раньше класс *Набор ряд* это подкласс *Набора*. Он представляет набор элементов с фиксированным диапазоном целых в качестве внешних ключей. В системе Смолток *Набор ряд* имеет пять подклассов: *Ряд*, *Цепь*, *Текст*, *Ряд серий* и *Ряд байтов*.

Ряд это набор чьи элементы это любые объекты. Он предоставляет конкретное представление для хранения набора элементов которые используют числа в качестве внешнего ключа. Несколько примеров использования *Рядов* уже было дано в этой главе.

Цепь это набор чьи элементы это *Знаки*. В этой и в предыдущих главах было дано много примеров использования *Цепей*. Класс *Цепь* предоставляет дополнительный протокол для инициализации и сравнения своих экземпляров.

Текст представляет *Цепь* у которой есть шрифт и выделение. В системе Смолток он используется для хранения информации нужной для создания текстовых документов. Экземпляры *Текста* имеют две переменные экземпляра, цепь и экземпляр *Ряда серий* в котором хранится изменение шрифта и выделения.

Класс *Ряд серий* предоставляет эффективный по памяти способ хранения данных которые не изменяются на больших интервалах индексов. Он запоминает повторяющийся элемент один раз и связывает с каждым элементом число которое показывает число после-

довательных вхождений элемента. Например, допустим что *Текст* представляет *Цепь* 'He is a good boy.' которая показывается со словом «boy» выделеным жирным шрифтом, и также допустим что код для шрифта это 1 и для его жирного варианта это 2. Тогда *Ряд серий* для *Текста* который связан с 'He is a good boy.' (*Цепь* из 17 знаков) содержит 1 связанную с 13, 2 связанную с 3, и 1 связанную с 1. Поэтому первые 13 *Знаков* имеют шрифт 1, следующие 3 шрифт 2, и последний знак шрифт 1.

Ряд байтов представляет *Набор ряд* чьи элементы это целые между 0 и 255. Реализация *Ряда байтов* запоминает два байта в 16-битных словах; класс поддерживает дополнительный протокол для доступа к словам и двойным словам. *Ряд байтов* используется в системе Смолток для хранения времени в миллисекундах.

10.6.1 Класс *Цепь*

Как было сказано выше, протокол класса для *Цепи* добавляет сообщения для создания копии другой *Цепи* (*из цепи: цепь*) или для создания *Цепи* из *Знаков* в *Потоке* (*читать из: поток*). Главное назначение второго сообщения в том что пары одинарных кавычек читаются и помещаются как один элемент, знак одинарная кавычка. Также, класс *Цепь* добавляет протокол сравнения подобно тому что определён для *Величины*. Некоторые из этих сообщений были введены раньше при описании класса *Сортированный набор*.

Протокол экземпляров *Цепи*

сравнение

< цепь
: Отвечает следует ли получатель перед аргументом, цепь. Сравнение производится в АСКОИ (ASCII) с игнорированием регистра.

<= цепь
: Отвечает следует ли получатель перед аргументом, цепь, или равен ей. Сравнение производится в АСКОИ (ASCII) с игнорированием регистра.

> цепь

Отвечает следует ли получатель за аргументом, цепь. Сравнение производится в АСКОИ (ASCII) с игнорированием регистра.

>= цепь
Отвечает следует ли получатель за аргументом, цепь, или равен ей. Сравнение производится в АСКОИ (ASCII) с игнорированием регистра.

сопоставить с: цепь
Рассматривает получатель как образец который содержить знаки # и *. Отвечает совпадает ли аргумент, цепь, с образцом получателя. Сопоставление игнорирует различие в регистре. Когда получатель содержит знак #, цепь может содержать любой один знак. Когда получатель содержит знак *, цепь может содержать любую последовательность знаков, в том числе ни одного.

такой же как: цепь
Отвечает равен ли словарно получатель аргумент, цепь. Сравнение производится в кодировке АСКОИ с игнорированием регистра.

До сих пор не было примеров использования последних двух сообщений.

предложение	**результат**
'first string' такой же как: 'first string'.	истина
'First String' такой же как: 'first string'.	истина
'First String' = 'first string'.	ложь
'#irst string' сопоставить с: 'first string'.	истина
'* string' сопоставить с: 'any string'.	истина
'*.st' сопоставить с: 'filename.st'.	истина
'first string' сопоставить с: 'first *'.	ложь

Цепь можно преобразовать к виду со всеми маленькими буквами или со всеми большими. Их также можно преобразовать в экземпляр класса *Символ*.

Протокол экземпляров *Цепи*

преобразование

в нижнем регистре
Возвращает *Цепь* созданную из получателя со всеми знаками в нижнем регистре.

в верхнем регистре
Возвращает *Цепь* созданную из получателя со всеми знаками в нижнем регистре.

как символ
Возвращает уникальный *Символ* чьи знаки это знаки получателя.

Поэтому получаем

предложение	**результат**
'first string' в верхнем регистре.	'FIRST STRING'
'First String' в нижнем регистре.	'first string'
'First' как символ.	First

10.6.2 Класс *Символ*

Символы это ряды *Знаков* для которых гарантируется уникальность. Поэтому:

'a string' как символ == 'a string' как символ.

возвратит истину. *Символ* предоставляет два сообщения для создания экземпляров в протоколе класса.

Протокол класса *Символ*

создание экземпляра

содержащий: **цепь**
Возвращает уникальный *Символ* чьи знаки такие же как у цепи.

содержащий знак: **знак**
Возвращает уникальный *Символ* из одного знака, аргумента знак.

В дополнение к этому, *Символы* можно создавать литерально, используя знак # как приставку к последовательности *Знаков*. Например, #dave это *Символ* из черытёх знаков. *Символы* печатаются без приставки.

10.7 Класс *Набор отображение*

Класс *Набор отображение* это подкласс *Набора*. Он представляет механизм доступа для ссылания на поднабор набора чьи элменты поименованы. Это отображение может определять перестановку или фильтр элементов набора. Основная идея состоит в том что *Набор отображение* ссылается на область и на карту. Область это *Набор* который используется для косвенного доступа через внешние ключи карты. Карта это *Набор* который связывает набор внешних ключей с другим множеством внешних ключей. Это второе множество ключей должно быть внешними ключами которые можно использовать для доступа к элементам области. Поэтому область и карта должны быть экземплярами *Словаря*, или его подкласса или подклассом *Набора последовательности*.

Возьмём, например, *Словарь* слов *Символов*, синонимы, введённый ранее.

ключ	значение
горячий	холодный
толкать	тянуть
стой	старт

приходи	иди
перед	зад
верх	низ

Допустим мы создали другой *Словарь* синонимов *Символов* для некоторых ключей антонимов и ссылаемся на их через альтернативные имена.

ключ	**значение**
прекрати	стой
входи	приходи
обжигающий	горячий
пихай	толкай

Замет можно создать Набор отображение выполнив предложение:

слова ← Набор отображение набор: антонимы карта: синонимы.

При помощи слов можно получить доступ к антонимам. Например, значение предложения слова от: #прекрати это старт (т.е. значение ключа прекрати в синонимах это стой, значение ключа стой в атонимах это старт). Можно определить какая часть антонимов доступна через слова при помощи сообщения содержимое.

слова содержимое.

Результат это *Мешок* содержащий символы старт, иди, холодный, тянуть.

Сообщение *от:пом:* это косвенный способ изменить набор область. Например:

предложение	**результат**
слова от: #обжигающий.	холодный
слова от: #прекрати.	старт
слова от: #прекрати пом: #продолжай.	продолжай
антонимы от: #стой.	продолжай

10.8 Краткое изложение преобразования между *Наборами*

В разделе описывающем различные виды наборов было указано что виды наборов могут преобразовываться друг в друга. Подводя итог, любой набор можно преобразовать в *Мешок*, *Множество*, *Упорядоченный набор* или *Сортированный набор*. Все наборы за исключением *Мешка* и *Множества* могут быть преобразованы в *Ряд* или *Набор отображение*. *Цепи* и *Символы* могут преобразовываться друг в друга, но ни один набор не может быть преобразован в *Интервал* или *Связанный список*.

Глава 11

Три примера использования наборов

Оглавление

В этой главе дано три примера описаний классов. Каждый пример использует объекты числа и набора доступные в системе Смолток, каждый пример показывает способ добавления функциональности в систему.

Карточную игру можно создать в терминах случайного выбора из набора представляющего колоду карт. Класс пример *Игралъная карта* представляет карту с какой-то мастью и достоинством. *Колода карт* представляет набор таких карт, *Карты на руках* это набор *Игралъных карт* некоторого игрока. Выбор карты из *Колоды карт* или из *Карт на руках* производится классами которы представляют выбор и замену, *Пространство выбора с заменой*, и без замены, *Пространство выбора без замены*. Хорошо известная задача программирования, задача пьяного таракана, включающая подсчёт количества шагов необходимых таракану случайно ползаю-

щему по всем плиткам в комнате. Решение данное в этой главе представляет каждую плитку как экземпляр класса *Плитка* и таракана как экземпляр класса *Пьяный таракан*. Третий пример из данной главы это дерево подобная структура данных представляемая классами *Дерево* и *Узел*, *Узел слово* показывает способ которым дерево можно использовать для хранения цепей представляющих слова.

11.1 Случайный выбор и игральные карты

Класс системы Смолток *Случайное число*, который работает как генератор случайно выбранного числа между 0 и 1, был описан в главе 8. Случайная величина предоставляет основу для выбора из множества возможных значений, такое множество называется пространством выборок. Простейшую форма дискретного случайного выбора можно получить используя случайное число для выбора элемента из последовательного набора. Если выбранный элемент остаётся в наборе, то выбор производится «с заменой» — то есть каждый элемент набора доступен при каждом выборе. В противоположность этому, выбираемый элемент может удаляться из набора при каждом выборе, это называется выбором «без замены».

Класс *Пространство выбора с заменой* предоставляет случайный выбор с заменой из последовательного набора элементов. Экземпляр класса создаётся заданием набора элементов из которого буду случайно выбираться элементы. Это сообщение инициализатор имеет селектор *данные:*. Выбор из набора осуществляется при помощи сообщения экземпляру с селектором *следующий*. Также можно получить целое количество выборок послав сообщение *следующий: целое*.

Например, допустим что нужно выбрать элементы из *Ряда Символов* представляющих имена людей.

```
люди ← Пространство выбора с заменой данные: #( #sally #sam
#sue #sarah #steve ).
```

Каждый раз когда нужно выбрать имя из *Ряда*, выполняется предложение.

```
люди следующий.
```

Ответ это один из *Символов*: sally, sam, sue, sarah или steve. Если выполнить предложение:

```
люди следующий: 5.
```

будет выбран *Упорядоченный набор* из пяти элементов. Экземпляры *Пространства выбора с заменой* отвечают на сообщения *пустой* и *размер* чтобы проверить есть ли элементы в пространстве выбора и сколько их. Поэтому ответ на:

```
люди пустой.
```

это ложь, и ответом на:

```
люди размер.
```

будет 5.

Далее дана реализация класса *Пространство выбора с заменой*. Для пояснения назначения метода в каждом методе дан комментарий, коментарии записываются в двойных кавычках.

имя класса Пространство выбора с заменой
надкласс Объект
имена переменных экземпляра данные случ

методы класса

создание экземпляра

данные: набор последовательность

```
"Создатёт экземпляр Пространства выбора с заменой такой что
аргумент, набор последовательность, является пространством выбо-
ра."
↑сам новый присвоить данные: набор последовательность.
```

методы экземпляра

доступ

следующий

```
  "Следующий элемент выбирается случайным образом из набора данные. Номер в наборе данные определяется при помощи случайного числа между 0 и 1, и его преобразования к диапазону набора данные."
  сам пустой истина: [сам ошибка: 'В пространтсве выбора нет значений'.].
  ↑данные от: (случ следующий * данные размер) усечь + 1.

проверки

пустой
  "Отвечает остались ли элементы для выбора."
  ↑сам размер = 0.
размер
  "Возвращает количество элементов оставшихся для выбора."
  ↑данные размер.

собственные

присвоить данные: набор последовательность
  "Аргумент, набор последовательность, это пространство выбора. Создаёт генератор случайных чисел для выбора из пространства."
  данные ← набор последовательность.
  случ ← Случайное число новый.
```

Описание класса указывает что у каждого экземпляра есть две переменные с именами данные и случ. Метод инициализации, *данные:*, посылает новому экземпляру сообщение *присвоить данные:* в котором данные связываются с *Набором последовательностью* (значением аргумента инициализирующего сообщения) а *случ* связывается с новым экземпляром класса *Случайное число*.

Пространство выбора с заменой не является подклассом *Набора* т.к. он реализует только малую часть сообщений на которые может отвечать *Набор*. В ответ на сообщения *следующий* и *размер* *Пространство выбора с заменой* посылает сообщения *от:* и *размер* переменной экземпляра *данные*. Это означает что любой набор отвечающий на сообщения *от:* и *размер* может быть использован в качестве данных из которые выбираются элементы. Все *Наборы*

последовательности отвечают на эти два сообщения. Поэтому, например, в дополнение к *Рядам* как было показано ранее, данные могут быть *Интервалом*. Допустим нужно смоделировать игральную кость. Значит элементы пространства выбора это положительные число от 1 до 6.

```
кость ← Пространство выбора с заменой данные: (1 до: 6).
```

Результат кидания кости определяется результатом предложения:

```
кость следующий.
```

Ответ этого сообщения это одно и целых: 1, 2, 3, 4, 5 или 6.

Можно выбирать карту из колоды таким же способом если набор связанный с экземпляров *Пространства выбора с заменой* содержит элементы представляющие игральные карты. Однако, чтобы играть в карточную игру, нужно работать с картами которые вынимаются из колоды и отдаются игроку. Поэтому здесь нужно использовать случайный выбор без замены.

Чтобы реализовать выбор без замены, нужно определить ответ на сообщение следующий с удалением выбранного элемента. Т.к. не все *Наборы последовательности* отвечают на сообщение *удалить:*, (например, *Интервал* не отвечает на это сообщение) нужно проверять аргумент в инициализирующем сообщении или преобразовывать аргумент к допустимому виду набора. Т.к. все *Упорядоченные наборы* отвечают на эти два сообщения, и т.к. все наборы можно преобразовать в *Упорядоченный набор*, то можно использовать такое преобразование. Чтобы выполнить преобразование метод связанный с *присвоить данные:* посылает своему аргументу сообщение *как упорядоченный набор*.

Класс *Пространство выбора без замены* определён как подкласс *Пространства выбора с заменой*. Методы связанные с сообщениями *следующий* и *присвоить данные:* переопределены, остальные сообщения наследуются без изменений.

имя класса Пространство выбора без замены

надркласс Пространство выбора с заменой

методы экземпляра

доступ

следующий
↑данные удалить: над следующий.

собственные

присвоить данные: набор
данные ← набор как упорядоченный набор.
случ ← Случайное число новый.

Заметьте что метод для селектора *следующий* зависит от метода реализованного в надклассе (*над следующий*). Метод надкласса проверяет не пусто ли пространство выбора и затем случайно выбирает элемент. После определения элемента, метод подкласса удаляет элемент из данных. Результат сообщения *удалить:* это аргумент, поэтому результат сообщения *следующий* посланного *Пространству выбора без замены* это выбранный элемент.

Давайте напишем простую карточную игру. Допустим что пространство выбора для карточной игры состоит из экземпляров класса *Игральная карта*, и каждая карта знает свою масть и достоинство. Экземпляр *Игральной карты* создаётся при помощи сообщения *масть:достоинство:*, определяющего с помощью двух аргументов масть (черви, буби, кресты, пики) и достоинство (1, 2, ..., 13). *Игральная карта* отвечает на сообщения масть и достоинство.

имя класса Игральная карта
надкласс Объект
имена переменных экземпляра масть достоинство

методы класса

создание экземпляра

масть: символ достоинство: целое
"Создаёт экземпляр Игральной карты чьи масть это аргумент, символ, и достоинство это аргумент, целое."

```
↑сам новый присвоить масть: символ достоинство: целое.
```

методы экземпляра

доступ

масть

```
"Возвращает масть получателя."
↑масть.
```

достоинство

```
"Возвращает достоинство получателя."
↑достоинство.
```

собственные

присвоить масть: символ достоинство: целое

```
масть ← символ.
достоинство ← целое.
```

Колода карт создаётся следующими предложениями.

```
колода карт ← Упорядоченный набор новый: 52.
#( #кресты #пики #буби #черви )
  делать: [
    :каждая масть |
    1
      до: 13
      делать: [
        :н |
        колода карт
              добавить: ( Игральная карта масть: каждая масть
достоинство: н ). ]. ].
```

Первое предложение создаёт *Упорядоченный набор* из 52-х элементов. Второе предложение это перечисление достоинств от 1 до 13 для каждой из четырёх мастей: кресты, пики, буби, черви. Каждый элемент *Упорядоченного набора* это *Игральная карта* с различным достоинством и мастью.

Можно получить возможность выбирать карты из колоды создав экземпляр *Пространства выбора без замены* с колодой карты в качестве набора из которого производится выбор.

карты ← Пространство выбора без замены данные: колода карт.

Чтобы выбрать карту нужно выполнить предложение:

карты следующий.

Ответом этого сообщения будет экземпляр класса *Игральная карта*.

Другой способ представления колоды игральных карт показан в описании класса-примера *Колода карт*. Основная идея заключается в хранении линейного списка карт; *следующий* означает первую карту в списке. Карту можно вернуть обратно в колоду поместив её в конец или вставив её в некоторую случайную позицию. Линейный список делается случайным при помощи тасования, т.е. при помощи случайного упорядочивания карт.

В реализации представленной ниже для *Колоды карт*, начальный вариант колоды карт запоминается в переменной класса. Она создаётся при помощи предложений данных выше. Копия этой переменной становится переменной экземпляра при создании нового экземпляра; она тасуется перед взятием карты. Каждое следующее тасование колоды использует текущее состояние переменной экземпляра, а не переменной класса. Процесс тасования, конечно, достаточно однообразен т.к. он основан на использовании экземпляра *Пространства выбора без замены*. Моделирование реалного тасования включает разделение колоды примерно пополам и затем перемешивание двух частей. Перемешивание включает выбор последовательностей из одной части и затем из другой части. Всё же, такое моделирование может быть более случайно чем тасование человеком, тасование человеком может быть предсказуемым и наблюдаемым.

Сообщения к *Колоде карт* с селекторами: *возвратить:*, *следующий* и *тасовать* полезны при создании карточной игры.

имя класса Колода карт
надркласс Объект
имена переменных экземпляра карты

имена переменных класса Начальная колода карт

методы класса

инициализация класса

инициализировать
```
  "Создаёт колоду из 52 игральных карт."
  Начальная колода карт ← Упорядоченный набор новый: 52.
  #( #буби #пики #кресты #черви )
    делать: [
      :масть |
      1
        до: 13
        делать: [
          :н |
          Начальная колода карт
            добавить: ( Игральная карта масть: масть достоинство:
н ). ]. ].
```

создание экземпляров

новый
```
   "Создаёт экземпляр Колоды карт с начальной колодой из 52–х
игральных карт."
  ↑ над новый карты: Начальная колода карт копия.
```

методы экземпляра

доступ

следующий
```
  "Выбирает слудеющую карту."
  ↑ карты удалить первый.
```
вернуть: карта
```
  "Помещает аргумент, карта, в конец колоды."
  карты добавить последним: карта.
```
тасовать
```
  | выбор временная колода |
```

```
  выбор ← Пространство выбора без замены данные: карты.
  временная колода ← Упорядоченный набор новый: карты размер.
  карты размер
    раз повторить: [временная колода добавить последним: выбор следующий.].
  сам карты: временная колода.

***проверки***

**пустой**
  "Отвечаеть есть ли ещё карты в колоде."
  ↑ карты пустой.

***собственные***

**карты: набор**
  карты ← набор.
```

Класс *Колода карт* нужно инициализировать выполнив предложение:

```
Колода карт инициализировать.
```

В реализации *Колоды карт*, карты это переменная экземпляра. Чтобы играть в карты создаётся экземпляр *Колоды карт*:

```
Колода карт новый.
```

и затем каждая карта вынимается при помощи сообщения *следующий* посылаемого данному новому экземпляру. Когда карта кладётся обратно в колоду, то экземпляру *Колоды карт* посылается сообщение *вернуть:*. Тасование всегда перемешивает карты находящиеся в колоде в данный момент. Если вся колода карт будет использоваться после завершение игры, то все карты взятые из колоды должны быть возвращены в колоду.

Обратите внимание на реализацию сообщения тасовать. *Пространство выбора без замены*, выбор, создаётся для копии текущей колоды карт. Новый *Упорядоченный набор*, временная колода, создаётся для хранения случайно выбранных карт из это пространства

выбора. Выбор из пространства производится для каждой возможной карты; каждая выбранная карта добавляется к временной колоде. Когда все доступные карты перемещены во временную колоду, она присваивается текущей колоде.

Допустим создаётся простая карточная игра с тремя или четырьмя игроками и раздающим. Раздающий раздаёт карты каждому игроку. Если хотябы один из игроков имеет от 18 до 21 очка, игра заканчивается с разделением «приза» между всеми такими игроками. Очки считаются суммированием достоинства карт. Игрок у которого больше чем 21 очко не получает новых карт.

Каждый игрок представляется при помощи экземпляра класса *Карты на руках*. *Карты на руках* знает карты которые он держит и общее количество очков для них (отвечает на сообщение *очки*).

имя класса Карты на руках
надкласс Объект
имена переменных экземпляра карты

методы класса

создание экземпляра

новый
↑над новый присвоить карты.

методы экземпляра

доступ

взять: **карта**
"Аргумент, карта, добавляется к получателю."
карты добавить: карта.
вернуть все карты в: **колода карт**
"Помещает все карты получателя в колоду на которую ссылается аргумент, колода карт, и удаляет эти карты из получателя."
карты делать: [**:каждая карта** | колода карт вернуть: каждая карта.].
сам присвоить карты.

запросы

очки
"Возвращает сумму достоинств карт получателя."
↑карты
ввести: 0
в: [**:значение :след карта** | значение + след карта достоинство.].

собственные

присвоить карты
карты ← Упорядоченный набор новый.

Создадим *Набор* из четырёх игроков. Каждый игрок это экземпляр *Карт на руках*. Карты раздающего это игральные карты. Раздающий (здесь, программист) начинает с тасования колоды; ещё нет победителя. Может быть больше одного победителя; победители будут перечисляться в *Наборе*, победители.

```
игроки ← Множество новый.
4 раз повторить: [игроки добавить: Карты на руках новый.].
игральные карты ← Колода карт новый.
игральные карты тасовать.
```

До тех пор пока нет победителя, каждый игрок у которого меньше чем 21 очко получает следующую карту из колоды карт. Перед раздачей карт всем подходящим игрокам, раздающий проверяет есть ли победители проверяя количество очков у каждого игрока.

```
[
 победители ← игроки выбрать: [:каждый | каждый очки между: 18 и: 21.].
  победители пустой и: [игральные карты пустой не.].]
    пока истина: [
      игроки
        делать: [
          :каждый |
          каждый очки < 21
           истина: [каждый взять: колода карт следующий.].].].
```

Условие продолжения игры это блок с двумя предложениями. Первое находит победителей, если они есть. Второе проверяет если если нету победителей (победители пустой) и, если их нет, есть ли ещё карты для раздачи (игральные карты пустой не). Если победителей нету и есть карты, то игра продолжается. Игра состоит из перебора игроков; каждый игрок получает карту (каждый взять: игральные карты следующий) если только количество очков меньше чем 21 (каждый очки $<$ 21). Чтобы сыграть ещё раз, все карты нужно вернуть в колоду, которая после этого тасуется.

```
игроки делать: [:каждый | каждый вернуть все карты в: игральные карты. ].
игральные карты тасовать.
```

Игроки и раздающий готовы к новой игре.

11.2 Задача о пьяном таракане

Можно использовать классы наборов для решения хорошо известной задачи программирования. Задача заключается в измерении времени требуемом пьяному таракану на посещение всех квадратных плиток на прямоугольном полу шириной H и длиной M. Немного идеализируем задачу: на текущем шаге таракан с одинаковой вероятностью переходит на одну из девяти плиток, т.е. на плитку на которой он находился до шага и плитки вокруг неё. Конечно движения таракана будут ограничены если таракан попытается выйти из комнаты. Задача сводится к подсчёту шагов требуемых таракану чтобы пройти по каждой плитке хотя бы по разу.

Один из прямых алгоритмов решения этой задачи начать с пустого *Множества* и счётчика установленного в ноль. После каждого шага, ко множеству добавляется плитка на которую перешёл таракан и увеличивается счётчик количества шагов. Т.к. в *Множестве* не допускается дублирование элементов алгоритм заканчивает работу когда число элементов множества достигнет значения H*M. При этом ответом будет значение счётчика.

Этот метод решает простейший вариант задачи, если дополнительно к этому нужно узнать некоторую дополнительную информацию, такую как — сколько раз была посещена каждая плитка.

Чтобы запомнить эту информацию, можно использовать экземпляр класса *Мешок*. Размер мешка это общее количество шагов совершённых тараканом, размер мешка после преобразования его во множество это общее количество плиток посещённых тараканом. Когда это число достигает значения H*M, решение задачи завершено. Количество вхождений каждой плитки в мешок это число посещений тараканом каждой плитки.

Каждуя плитку на полу можно пометить в соответствии с её рядом и столбцом. Объекты представляющие плитки будут экземплярами класса *Плитка*. Реализация класса *Плитка*:

имя класса Плитка
надркласс Объект
имена переменных экземпляра положение область пола

методы экземпляра

доступ

положение
"Возвращает положение получателя на полу."
↑ положение.
положение: точка
"Присваивает положению получателя аргумент, точка."
положение ← точка.
область пола: прямоугольник
"Присваивает области пола аргумент — прямоугольную область, прямоугольник."
область пола ← прямоугольник.

перемещение

ближайший к: точка дельта
"Создаёт и возвращает новую Плитку находящуюся в положении получателя изменённом на икс и игрек аргумента, точка дельта. Новая плитка остаётся в пределах границ области пола."
| новая плитка |
новая плитка ← Плитка новый область пола: область пола.

```
  новая плитка
      положение: ((положение + точка дельта макс: область пола
начало) мин: область пола угол).
  ↑ новая плитка.

сравнение

= плитка
  "Отвечает равен ли получатель аргументу, плитка."
  ↑ (плитка это разновидность: Плитка) и: [положение = плитка
положение.].
хэш
  ↑ положение хэш.
```

Плитка ссылается на свой ряд и столбец, каждый из которых должен быть больше либо равен 1 и не больше ширины и высоты пола. Поэтому в дополнение к положению плитка должна помнить область пола в которой она может быть помещена. Плитке можно послать сообщение ближайший к: точка чтобы определить плитку наиболее близко лежащую к этой точке. Эта новая плитка должна находится в пределах области пола.

Способ которым будет моделироваться движение таракана это выбор направления в терминах изменения координат таракана икс и игрек. Для данного положения таракана (плитка икс, игрек) есть 9 плиток на которые насекомое может переползти если плитка не находится у границы. Возможные изменения координат будут храниться в *Упорядоченном наборе* который является данными для случайного выбора. *Упорядоченный набор* будет содержать в качестве элементов *Точки*, точки это векторы направления представляющие все возможные комбинации движений. Этот набор создаётся предложением:

```
Направления ← Упорядоченный набор новый: 9.
(−1 до: 1)
  делать: [
    :икс |
  (−1 до: 1) делать: [:игрек | Направления добавить: икс @ игрек.].].
```

Поэтому *Направления* это набор с элементами:
-1@-1, -1@0, -1@1, 0@-1, 0@0, 0@1, 1@-1, 1@0, 1@1

Как части задачи пьяного таракана, нужно будет генерировать случайные числа для выбора элемента из *Упорядоченного набора* возможных движений. Как альтернативный вариант прямого использования генератора случайных числе, можно использовать предыдущий пример *Пространство выбора с заменой* и *Направления* как пространство выбора.

Допустим таракан начинает двигаться с плитки с координатами 1 @ 1. Каждый раз когда таракан делает шаг выбирается элемент из набора *Направления*. Этот элемент затем передаётся в качестве аргумента сообщения плитке, *ближайший к:*. Дальше предполагается что *Случ* это экземпляр класса *Случайное число*.

```
плитка
 ближайший к: ( Направления от: ( Случ следующий * Направления размер ) усечь + 1 ).
```

Плитка результат это место где находится таракан.

Каждая плитка положение запоминается чтобы иметь информацию о том посещены ли все плитки и сколько на это потребовалось шагов. Если помещать каждую плитку в *Мешок*, то дубликаты будут указывать количество посещений каждого положения. Поэтому на каждом шаге создаётся копия плитки с предыдущего шага. Эта новая плитка изменяется в соответствии со случайно выбранным направлением и она добавляется в *Мешок*. Когда количество уникальных элементов в *Мешке* достигает общего количества плиток, решение задачи завершается.

В классе *Пьяный таракан* нужно только два сообщения. Одно сообщение это команда двигаться по заданной области до тех пор пока не будут посещены все плитки. Это сообщение *ходить по:начиная с:*. Второе сообщение это запрос о общем количестве шагов выполненных тараканом, это сообщение *количество шагов*. Также можно запросить количество посещений конкретной плитки послав *Пьяному таракану* сообщение *количество посещений:*. Набор векторов направлений (как было описано выше) создаётся как переменная класса *Пьяного таракана*, генератор случайных чисел *Случ* также является переменной класса Пьяного таракана.

имя класса Пьяный таракан
надркласс Объект
имена переменных экземпляра текущая плитка посещённые плитки
имена переменных класса Направления Случ

методы класса

инициализация класса

инициализировать
"Создаёт набор векторов направлений и генератор случайных чисел."
Направления ← Упорядоченный набор новый: 9.
(−1 до: 1)
делать: [
:икс |
(−1 до: 1) делать: [**:игрек** | Направления добавить: икс @ игрек.].].
Случ ← Случайное число новый.

создание экземпляров

новый
↑ над новый присвоить переменные.

методы экземпляра

моделирование

ходить по: прямоугольник начиная с: точка
| количество плиток |
посещённые плитки ← Мешок новый.
текущая плитка положение: точка.
текущая плитка область пола: прямоугольник.
количество плиток ← прямоугольник ширина + 1 * (прямоугольник высота + 1).
посещённые плитки добавить: текущая плитка.
[
посещённые плитки как множество размер

```
    < количество плиток. ]
    пока истина: [
      текущая плитка ← текущая плитка
        ближайший к: ( Направления
          от: ( Случ следующий * Направления размер )
            усечённый
              + 1 ).
      посещённые плитки добавить: текущая плитка. ].
```

данные

количество шагов
↑ посещённые плитки размер.
количество посещений: плитка
↑ посещённые плитки вхождений: плитка.

собственные

присвоить переменные
текущая плитка ← Плитка новый.
посещённые плитки ← Мешок новый.

Сейчас можно послать следующие сообщения чтобы поэкспериментировать с пьяным тараканом. Инициализируем класс и создадим экземпляр.

```
Пьяный таракан инициализировать.
таракан ← Пьяный таракан новый.
```

Получение результатов 10-ти экспериментов с комнатой 5 на 5.

```
результаты ← Упорядоченный набор новый: 10.
10
  раз повторить: [
    таракан ходить по: ( 1 @ 1 угол: 5 @ 5 ) начиная с: 1 @ 1.
    результаты добавить: таракан количество шагов. ].
```

Среднее этих 10-ти значений это среднее значение шагов требуемых пьяному таракану на решение задачи.

```
( результаты ввести: 0 в: [ :сум :эксп | сум + эксп. ] ) / 10.
```

Заметьте что в реализации сообщения *Пьяного таракана ходить по:начиная с:* условие завершения это достижние количества элементов в множестве преобразованном из мешка величины H*M. Более быстрый способ проверить это условие — добавить сообщение *уникальных элементов* в класс *Мешок* чтобы не требовалось производить преобразования во *Множество* на каждой итерации.

(Для тех читателей которые хотят попробовать это добавление, метод который нужно добавить в класс *Мешок*:

уникальных элементов
↑содержимое размер.

Затем сообщение *ходить по:начиная с:* можно изменить так чтобы условие завершения было таким: посещённые плитки уникальных элементов < количество плиток.)

11.3 Обход бинарного дерева

Дерево это важная нелинейная структура данных которая полезна в компьютерных алгоритмах. Древовидная структура это такая структура в которой отношения между элементами это ветви. Есть один элемент называемый корнем дерева. Если существует только один элемент, то он является корнем. Если есть дополнительные элементы, то они разделяются на непересекающиеся поддеревья. Бинарное дерево это либо только корень, корень и одно бинарное поддерево, или корень с двумя бинарными поддеревьями. Полное описание древовидных структур есть в первом томе Искусства программирования Кнута. Здесь предполагается что читатель знаком с идеей деревьев поэтому здесь показывается как определить эту структуру данных как класс системы Смолток.

Мы определим класс *Дерево* способом подобным определению класса *Связанный список*. Элементами *Дерева* будут *Узлы* которые подобно *Связям Связанного списка* могут соединяться с другими элементами. *Дерево* ссылается только на корень.

Узел представляется как объект Смолтока с двумя переменными экземпляра, одна ссылается на левый узел а вторая на правый узел. Выбран симметричный порядок обхода дерева. Это значит что при переборе узлов сначала просматривается левое поддерево, вторым

корень, и третьим правое поддерево. Если узел не имеет поддеревьев, то он называется листом. По определению размер узла это 1 плюс размер его поддеревьев, если они есть. Поэтому лист это узел размера 1, и узел с двумя листами имеет размер 3. Размер дерева это размер его корня. Это определение размера соответствует общему определению размера для наборов.

Сообщения *Узла* дают доступ к левому узлу, правому узлу, и к последнему узлу. Также возможно удалить подузел (*удалитъ:если нету:*) и корень (*остаток*).

имя класса Узел
надркласс Объект
имена переменных экземпляра левый узел правый узел

методы класса

создание экземпляра

левый: лев узел правый: прав узел
"Создаёт экземпляр Узла с аргументами л узел и п узел как левым и правым подузлом соответственно."
| новый узел |
новый узел ← сам новый.
новый узел левый: лев узел.
новый узел правый: прав узел.
↑ новый узел.

методы экземпляра

проверки

это лист
"Отвечает является ли получатель листом, т.е. отсутствуют ли у узла подузлы."
↑ левый узел это пусто & правый узел это пусто.

доступ

левый

```
  ↑ левый узел.
левый: узел
  левый узел ← узел.
правый
  ↑ правый узел.
правый: узел
  правый узел ← узел.
размер
  ↑ 1 + ( левый узел это пусто истина: [ 0. ] ложь: [ левый узел размер. ] )
      + ( правый узел это пусто истина: [ 0. ] ложь: [ правый узел размер. ] ).
последний
  | узел |
  узел ← сам.
  [ узел правый это пусто. ] пока ложь: [ узел ← узел правый. ].
  ↑ узел.
```

удаление

```
удалить: подузел если нету: блок исключение
  "Предполагается что корень, сам, нельзя удалить."
  сам это лист истина: [ ↑ блок исключение значение. ].
  левый узел = подузел истина: [ левый узел ← левый узел остаток.
↑ подузел. ].
  правый узел = подузел
    истина: [ правый узел ← правый узел остаток. ↑ подузел. ].
  левый узел это пусто
    истина: [ ↑ правый узел удалить: подузел если нету: блок исключение. ].
  ↑ левый узел
    удалить: подузел
    если нету: [
      правый узел это пусто
        истина: [ ↑ блок исключение значение. ]
        ложь: [ правый узел удалить: подузел если нету: блок исключение. ]. ].
```

```
остаток
  левый узел это пусто
    истина: [↑правый узел.]
   ложь: [левый узел последний правый: правый узел. ↑левый узел.].
```

перебор

```
делать: блок
  левый узел это пусто ложь: [левый узел делать: блок.].
  блок значение: сам.
  правый узел это пусто ложь: [правый узел делать: блок.].
```

Перебор использует симметричный обход, сначала в качестве значения для блока применяется левый подузел, затем корень, и третьим правый подузел. Блок должен быть определён для аргумента являющегося *Узлом*.

Представим *Дерево* как вид *Набора последовательности* чьи элементы это *Узлы*. *Дерево* имеет одну переменную экземпляра с именем корень, корень это пусто или экземпляр *Узла*. Как подкласс *Набора последовательности*, класс *Дерево* реализует сообщения *добавить: элемент*, *удалить: элемент если нету: блок исключение*, и *делать: блок*. В основном эти методы проверяют пустое ли дерево (корень это пусто) и, если нет, посылают соответствующее сообщение корню. Проверка на пустоту наследуется от *Набора*. Это сделано для того чтобы программист использующий древовидную структуру обращался к элементам этой структуры только через экземпляры класса *Дерево*.

```
имя класса Дерево
надркласс Набор последовательность
имена переменных экземпляра корень
```

методы экземпляра

проверки

```
пустой
  ↑корень это пусто.
```

доступ

первый
| узел |
сам проверить на пустоту.
узел ← корень.
[узел левый это пусто.] пока ложь: [узел ← узел левый.].
↑ узел.
последний
сам проверить на пустоту.
↑ корень последний.
размер
сам пустой истина: [↑ 0.] ложь: [↑ корень размер.].

добавление

добавить: узел
↑ сам добавить последним: узел.
добавить первым: узел
"Если набору пусто, тогда аргумент, узел, это корень; иначе, это левый узел текущего первого узла."
сам пустой истина: [↑ корень ← узел.] ложь: [сам первый левый: узел.].
↑ узел.
добавить последним: узел
"Если набор пусто, то аргумент, узел, это новый корень; иначе это правый узел текущего последнего узла."
сам пустой истина: [корень ← узел.] ложь: [сам последний правый: узел.].
↑ узел.

удаление

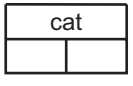
удалить: узел если нету: блок исключение
"Сначала проверяется корень. Если узел не равен корню, то просматривается всё дерево."
сам пустой истина: [↑ блок исключение значение.].
корень = узел

```
    истина: [корень ← корень остаток. ↑узел.]
    ложь: [↑корень удалить: узел если нету: блок исключение.].
удалить первый
  сам проверить на пустоту.
  ↑сам удалить: сам первый если нету: [].
удалить последний
  сам проверить на пустоту.
  ↑сам удалить: сам последний если нету: [].

перебор

делать: блок
  сам пустой ложь: [корень делать: блок.].
```

Заметьте что удаляющее сообщение не удаляет поддерево начинающиеся с узла, а только сам узел.

11.3.1 Бинарное дерево слов

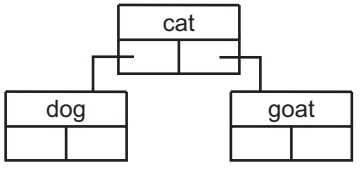
Определение *Узла*, подобно *Связи*, это структура без содержания. Содержание каждого узла определяется подклассом. Допустим нужно использовать вид *Узла* для хранения слов. Назовём этот класс *Узел слово*. Экземпляр *Узла слова* создаётся при помощи сообщения *для:*, если не задаются подузлы, или *для:правый:левый:* если задаётся два подузла. Поэтому *Узел слово* показываемое как:



создаётся при выполнение предложения

```
Узел слово для: 'cat'.
```

Узел слово выглядищее так:



создаётся при выполнении предложения:

```
Узел слово
  для: 'cat'
  левый: ( Узел слово для: 'dog' )
  правый: ( Узел слово для: 'goat' ).
```

Ниже приведена реализация класса *Узел слово*. Заметьте что равенство (=) переопределено так чтобы означать равенство слов в *Узле*, это означает что унаследованное удаляющее сообщение будет удалять подузел если слово в нём такое же как в аргументе.

имя класса Узел слово
надркласс Узел
имена переменных экземпляра слово

методы класса

создание экземпляра

```
для: цепь
  ↑ сам новый слово: цепь.
для: цепь левый: лев узел правый: прав узел
  | новый узел |
  новый узел ← над левый: лев узел правый: прав узел.
  новый узел слово: цепь.
  ↑ новый узел.
```

методы экземпляра

доступ

```
слово
  ↑ слово.
слово: цепь
  слово ← цепь.
```

сравнение

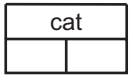
```
= узел слово
  ↑ ( узел слово это разновидность: Узел слово ) и: [ слово = узел слово слово. ].
```

хэш

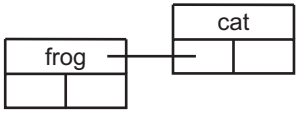
↑слово хэш.

Селдующая последовательность предложений иллюстрирует использование *Узла слова*. Заметьте что в определении *Узла слова* не было сделано ничего для поддержки вставки элементов так чтобы при обходе дерева они располагались в алфавитном порядке. Заинтересованный читатель может добавить сообщение *вставить: узел слово* в *Узел слово* которое сохраняет упорядоченность по алфавиту.

```
дерево ← Дерево новый.

дерево добавить: ( Узел слово для: 'cat' ).
```

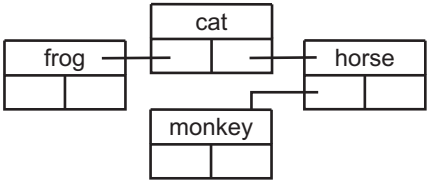



```
дерево добавить первым: ( Узел слово для: 'frog' ).
```

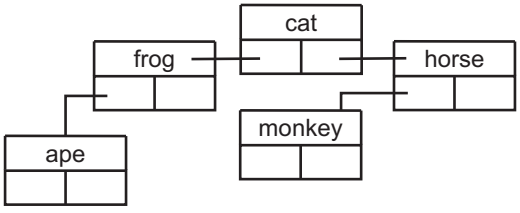



```
дерево
```

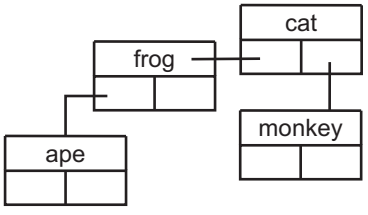
добавить последним: (Узел слово для: 'horse' левый: (Узел слово для: 'monkey') правый: пусто).



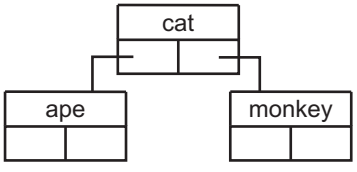

```
дерево добавить первым: ( Узел слово для: 'ape' ).
```




```
дерево удалить: ( Узел слово для: 'horse' ).
```




```
дерево удалить: ( Узел слово для: 'frog' ).
```



Глава 12

Протокол потоков

Оглавление

Классы наборов предоставляют основные структуры данных для хранения объектов в виде линейных и нелинейных групп. Протокол этих классов позволяет напрямую обращаться (сохранять и получать) к отдельным элементам. Также, через перечисляющие сообщения, поддерживается непрерывный доступ ко всем объектам, по порядку. Однако, они не поддерживают смешивания двух видов операций доступа — перечисление и запоминание. Также протокол наборов поддерживает доступ к индивидуальным элементам, к одному за раз, потому что внешняя позиция хранится отдельно.

Если не существует легко вычисляемых внешних имён для каждого элемента, произвольное перечисление элементов нельзя произвести эффективно. Например, возможно последовательное чтение

Объект

Величина
Знак
Дата
Время

Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое

Ключ поиска
Ассоциация

Связь
Процесс

Набор
Набор последовательность
Связанный список

Семафор

Набор ряд
Ряд

Растровое изображение

Ряд серий
Цепь
Символ
Текст
Ряд байтов

Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь

Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл

Случайное число

Неопределённый объект
Логика
Истина
Ложь

Планировщик исполнителя
Задержка
Разделяемая очередь

Поведение
Описание класса
Класс
Метакласс

Точка
Прямоугольник

элементов *Упорядоченного набора* используя комбинацию сообщений *первый* и *после:*, до тех пор пока элементы набора уникальны. Альтернативный подход использует запоминание самим набором к которому элементу было последние обращение. Это называется «ссылка на положение» в последующем обсуждении. Однако, возможность раздельного доступа к последовательности элементов означает что нужно хранить отдельно последний использованный элемент.

Класс *Поток* предоставляет возможность сохранять ссылку на положение в наборе объектов. Фраза поток через набор означает доступ к элементам набора таким образом что возможно перечисление или запоминание каждого элемента, одного за раз, с возможным перемешиванием этих операций. Создавая несколько *Потоков* для одного набора, можно управлять многими ссылками на положение в одном и том же наборе.

Существует несколько способов хранить ссылку на положения для потока через набор. Один из частых способов это целый номер. Этот подход может быть использован для любого набора чьи элементы именуются снаружи при помощи целых. Все *Наборы последовательности* попадают в эту категорию. Как будет показано позже, такие *Потоки* представляются в системе Смолток классом *Позиционируемый поток*. Второй способ управлять ссылком положением это использование затравки для генератора объектов. Примером данного вида *Потока* в системе Смолток является класс *Случайное число* который уже был представлен в восьмой главе. И третий способ это использование не числового положения ссылки, такой как ссылка на узел в последовательности, данный подход иллюстрируется в этой главе примером класса который поддерживает поток через связанный список или древовидную структуру.

12.1 Класс *Поток*

Класс *Поток*, подкласс *Объекта*, это надкласс который определяет протокол доступа для потоков над наборами. В этот протокол включены сообщения для чтения и записи в набор, однако не все подклассы *Потока* могут поддерживать оба вида операций доступа. Основное сообщение чтения это следующий, оно возвращает

следующий элемент набора на который ссылается *Поток*. Получив доступ к следующему элементу, можно создать более общие сообщения чтения. Такими сообщениями являются *следующий: целое*, которое возвращает набор из целоего количества элементов; *следующий совпадает с: объект*, которое читает следующий элемент и отвечает равен ли он аргументу, объект; и *содержимое*, которое возвращает набор всех элементов.

Протокол экземпляров *Потока*

доступ-чтение

следующий
Возвращает следующий объект доступный получателю.

следующий: целое
Возвращает следующие целое количество объектов доступных получателю. Как правило, ответом будет набор того же класса что и набор получателя.

следующий совпадает с: объект
Получает следующий объект и отвечает равен ли он аргументу, объект.

содержимое
Возвращает все объекты в наборе доступном получателю. Как правило, ответом будет набор того же класса что и набор получателя.

Основное сообщение записи это *пом следующим: объект*; оно означает помещение аргумента, объект, в качестве следующего элемента доступного получателю. Если возможны и сообщения чтения и сообщения записи, то сообщение *следующий* выполненное после *пом следующим: объект* не должно прочитать только что запомненный элемент, а вместо этого должно прочитать следующий за ним элемент в наборе. Сообщения записи также включают *пом следующими все: набор*, которое запоминает все элементы аргумента в набор доступный получателю, и *следующими: целое пом: объект*, которое запоминает аргумент, объект, целое количество раз.

Протокол экземпляров *Потока*

доступ-запись

пом следующим: **объект**
Запоминает аргумент, объект, как следующий элемент доступный получателю. Возвращает объект.

пом следующими все: **набор**
Запоминает элементы аргумента, набор, как следующие элементы доступные получателю. Возвращает набор.

следующими: **целое** **пом:** **объект**
Запоминает аргумент, объект, как следующий элемент целое число раз. Возвращает объект.

Сообщения чтения и записи определяют можно ли прочитать или записать следующий элемент, и если нельзя, то сообщается об ошибке. Поэтому программист может захотеть определить когда доступ ещё возможен; это достигается посылкой *Потоку* сообщения в конце.

Протокол экземпляров *Потока*

проверки

в конце
Отвечает не может ли получатель вернуть объекты из набора.

Непрерывное чтение элементов, которые применяются как аргумент к блоку, можно произвести послав сообщение *делать: блок*, подобно перебирающему сообщению поддерживаемому классами наборов.

Протокол экземпляров *Потока*

перебор

делать: **блок**
Выполняет аргумент, блок, для каждого оставшегося элемента который доступен получателю.

Реализация этого перечисляющего сообщения зависит от сообщений *в конце* и *следующий*. Покажем этот метод как пример использования этих сообщений.

```
делать: блок
  [сам в конце.] пока ложь: [блок значение: сам следующий.].
```

Каждый вид *Потока* должен определять сообщения создания своего экземпляра. *Поток* не может просто создавать экземпляры при помощи сообщения *новый*, т.к. *Поток* должен знать к которому набору он имеет доступ и чему равна начальная ссылка.

В качестве простого примера, предположим что набор доступный *Потоку* это *Ряд* и что *Поток* называется доступ. Содержимое это *Ряд* из *Символов*:
Bob Dave Earl Frank Harold Jim Kim Mike Peter Rick Sam Tom
и ссылка такая что следующий доступный элемент это Bob. Тогда:

предложение	**результат**
доступ следующий.	Bob
доступ следующий.	Dave
доступ следующий совпадает с: #Bob.	ложь
доступ следующий совпадает с: #Frank.	истина
доступ следующий.	Harold
доступ пом следующим: #James.	James
доступ содержимое.	(Bob Dave Earl Frank Harold James Kim Mike Peter Rick Sam Tom)
доступ пом следующим все: #(#Karl #Larry #Paul).	(Karl Larry Paul)
доступ содержимое.	(Bob Dave Earl Frank Harold James Karl Larry Paul Rick Sam Tom)
доступ следующими: 2 пом: #John.	John

доступ содержимое.	(Bob Dave Earl Frank Harold James Karl Larry Paul John John Tom)
доступ следующий.	Tom
доступ в конце.	истина

12.2 Позиционируемый поток

Во введении к этой главе было указано три возможных подхода используемых потоками для управления ссылкой на положение. Первый который будет рассмотрен это целый номер который увеличивается каждый раз при доступе к потоку. Поток использует только такие виды наборов для который внешний ключ это целые числа, в этот вид входят все подклассы *Набора последовательности*.

Класс *Позиционируемый поток* это подкласс класса *Поток*. Он предоставляет дополнительный протокол подходящий для потоков которые могут изменять свою ссылку положение, но, это абстрактный класс т.к. он не предоставляет реализации унаследованных сообщений *следующий* и *следующим пом: объект*. Реализация этих сообщений оставлена подклассам *Позиционируемого потока* — *Потоку чтения*, *Потоку записи* и *Потоку чтения записи*.

Позиционируемый поток создаётся при помощи посылки классу одного из двух сообщений создания экземпляра, *на: набор* или *на: набор от: первый номер до: последний номер*. Аргумент, *набор*, это набор доступный потоку; во втором случае доступна копия поднабора, т.е. поднабор ограниченный двумя аргументами *первый номер* и *последний номер*.

Протокол класса *Позиционируемый поток*

создание экземпляра

на: **набор**
Возвращает экземпляр вида *Позиционируемого потока* который протекает через аргумент, набор.

на: набор от: первый номер до: последний номер
Возвращает экземпляр вида *Позиционируемого потока* который протекает через копию поднабора аргумента, набор, от первого номера до последнего номера.

Позиционируемый поток поддерживает дополнительный протокол для доступа и проверки содержимого набора.

Протокол экземпляров *Позиционируемого потока*

проверки

пустой
Отвечает истина если набор доступный получателю не имеет элементов, иначе возвращает ложь.

доступ

считать
Возвращает следующий элемент из набора (как и в сообщении следующий), но не изменяет ссылку на положение. Возвращает пусто если находится в конце.

считать для: объект
Определяет ответ на сообщение считать. Если это то же что и аргумент, объект, то увеличивает ссылку на положение и возвращает истину. Иначе возвращает ложь и не изменяет ссылку на положение.

вплоть до: объект
Возвращет набор элементов начинающийся со следующего элемента доступного получателю, и до, но не включая, следующего элемента который равен объекту. Если объекта нету в наборе, то возвращается весь остаток набора.

Т.к. для *Позиционируемого потока* известно что он хранит ссылку на положение, то поддерживается протокол для доступа к этой ссылке. В частности ссылка может быть установлена на начало, на конец, или на любое другое положение набора.

Протокол экземпляров *Позиционируемого потока*

позиционирование

позиция
Возвращает текущую ссылку положение получателя для доступа к набору.

позиция: **целое**
Присваивает текущуей ссылке положению получателя для доступа к набору аргумент, целое. Если аргумент аргумент выходит за пределы набора получателя, то сообщается об ошибке.

сбросить
Присваивает ссылке положению получателя начало набора.

установить в конец
Присваивает ссылке положению получателя конец набора.

пропустить: **целое**
Присваивает ссылке положению получателя текущую позицию плюс аргумент, целое, возможно с подгонкой результата так чтобы он остался в пределах границ набора.

пропустить до: **объект**
Присваивает ссылке положению получателя позицию перед следующим вхождением в набор аргумента, объект. Отвечает существует ли такое вхождение.

12.2.1 Класс *Поток чтения*

Класс *Поток чтения* это конкретный подкласс *Позиционируемого потока* который представляет объект доступа который может только читать элементы из набора. Для демонстрации использования дополнительного протокола предоставленного классом *Позиционируемый поток* и наследуемого *Потоком чтения* можно написать пример подобный предыдущему. Заметьте что сообщения *пом следующим:*, *следующими:пом:* и *пом следующими все:* не могут быть успешно посланы экземпляру *Потока чтения*.

```
доступ ← Поток чтения
  на: #(
    #Bob
    #Dave
    #Earl
```

```
#Frank
#Harold
#Jim
#Kim
#Mike
#Peter
#Rick
#Sam
#Tom ).
```

предложение	результат
доступ следующий.	Bob
доступ следующий совпадает с: #Dave.	истина
доступ считать.	Earl
доступ следующий.	Earl
доступ считать для: #Frank.	истина
доступ следующий.	Harold
доступ вплоть до: #Rick.	(Jim Kim Mike Peter)
доступ позиция.	10
доступ пропустить: 1.	сам доступ
доступ следующий.	Tom
доступ в конце.	истина
доступ сбросить.	сам доступ
доступ пропустить до: #Frank.	истина
доступ следующий.	Harold

12.2.2 Класс *Поток записи*

Класс *Поток записи* это подкласс *Позиционируемого потока* предоставляющего доступ для записи элементов в набор. Ни одно из сообщений *следующий*, *следующий:* и *делать:* нельзя успешно послать *Потоку записи*.

Поток записи используется во всей системе Смолток как часть методов печати или помещения строки описывающей любой объект. Каждый объект системы может отвечать на сообщения *печатать*

в: поток и *поместить в: поток*. Методы связанные с этими сообщениями содержат последовательность сообщений аргументу, который является видом *Потока* позволяющим записывать элементы в доступный ему набор. Эти сообщения это *пом следующим:*, с аргументом *Знаком*; и *пом следующими все:*, с аргументом *Цепью* или *Символом*. Проиллюстрируем эту идею примером.

Как описано в главе 6, протокол печати класса *Объект* включает сообщение *цепь для печати*. Реализация этого сообщения:

```
цепь для печати
  | поток |
  поток ← Поток записи на: ( Цепь новый: 16 ).
  сам печатать в: поток.
  ↑ поток содержимое.
```

Если набору посылается сообщение цепь для печати, то ответом будет *Цепь* которая описывает экземпляр. Метод создаёт *Поток записи* в который будет помещаться набор, набору посылается сообщение *печатать в:*, и затем возвращается содержимое результирующего *Потока записи*. Сообщение *цепь для помещения* реализовано для всех объектов в классе *Объект* подобным образом, различие заключается в том что во втором предложении используется сообщение *поместить в: поток* вместо сообщения *печатать в: поток*.

Общий способ которым наборы печатают своё описание это печать имени своего класса, последующей открывающейся скобки, печать описаниея каждого элемента с разделением пробелом, и завершающей закрывающейся скобки. Поэтому если *Множество* имеет четыре элемента — символы один, два, три и четыре — тогда оно будет напечатано в поток так:

Множество (один два три четыре)

Упорядоченный набор с теми же элемента напечатается в поток так:

Упорядоченный набор (один два три четыре)

и т.д.

Напомним что определение *печатать в:* и *поместить в:* данное в главе 6 такое что любое подходящее описание может быть предоставлено сообщением *печатать в:*, но описание созданное со-

общение *поместить в:* должно быть правильно сформированным предложением таким чтобы при выполнении воссоздать описываемый объект.

Вот реализации сообщения *печатать в:* класса *Набор*.

```
печатать в: поток
  поток пом следующими все: сам класс имя.
  поток пробел.
  поток пом следующим: $(.
  сам делать: [:элемент | элемент печатать в: поток. поток пробел.].
  поток пом следующим: $).
```

Обратите внимание на сообщение *пробел* посылаемое *Потоку записи* (*поток*). Это и несколько других сообщений предоставляются классом *Поток записи* для поддержки кратких выражений помещающих разделители в *Потоки*. В эти сообщения включаются:

Протокол экземпляров *Потока записи*

запись знаков

пс
: Помещает знак перевода строки в качестве следующего элемента получателя.

пс таб
: Помещает знак перевода строки и одну табуляцию в качестве следующих элементов получателя.

пс таб: целое
: Помещает знак перевода строки в качестве следующего элемента получателя, за которым следует целое число табуляций.

пробел
: Помещает знак пробела в качестве следующего элемента получателя.

таб
: Помещает знак табуляции в качестве следующего элемента получателя.

Поэтому чтобы составить *Цепь*:

name	city
bob	New York
joe	Chicago
bill	Rochester

из двух соответствующих *Рядов*,

```
имена ← #( #bob #joe #bill ).
города ← #( 'New York' 'Chicago' 'Rochester' ).
```

выполните предложения:

```
поток ← Поток записи на: ( Цепь новый: 16 ).
поток пом следующими все: 'name'.
поток таб.
поток пом следующими все: 'city'.
поток пс; пс.
имена
    с: города
    делать: [
        :имя :город |
        поток пом следующими все: имя.
        поток таб.
        поток пом следующими все: город.
        поток пс. ].
```

после этого желаемый результат получается при выполнении *поток содержимое*.

Допустим что набор уже существует и нужно добавить в него ещё некоторую информацию используя протокол *Потока*. Класс *Поток записи* поддерживает протокол создания экземпляров который берёт набор и устанавливает ссылку на положение записи в конец.

Протокол класса *Поток записи*

создание экземпляров

с: набор

Возвращает экземпляр *Потока записи* имеющего доступ к аргументу, набор, но установленный так чтобы записать следующий элемент в его конец.

с: набор от: первый номер до: последний номер
Возвращает экземпляр *Потока записи* которому доступен поднабор аргумента, набор, от первого номера до последнего номера, но установленный так чтобы записать следующий элемент в конец поднабора.

Поэтому если *Цепь* с именем *заголовок* уже существует и содержит:

```
name city
```

тогда предыдущий пример нужно строить так:

```
поток ← Поток записи с: заголовок.
имена
  с: города
  делать: [
    :имя :город |
    поток пом следующими все: имя.
    поток таб.
    поток пом следующими все: город.
    поток пс. ].
поток содержимое.
```

12.2.3 Класс *Поток записи чтения*

Класс *Поток записи чтения* это подкласс *Потока записи* который представляет из себя объект доступа который может и читать и записывать элементы в свой набор. Он поддерживает оба протокола и *Потока чтения* и *Потока записи*, которые были даны выше.

12.3 Потоки генерируемых элементов

Из трёх способов хранения ссылки на положение для протекания через набор, упомянутых в начале этой главы, вторым способом

было использование затравки по которой создаётся следующий элемент набора. Этот вид потока может только читать элементы, но не писать их. Однако ссылка может быть перенастроена при помощи изменения затравки.

Класс *Случайное число*, введённый в главе 8, это подкласс *Потока* который находит свои элементы на основе алгоритма использующего число в качестве затравки. *Случайное число* предоставляет конкретную реализацию для сообщений *следующий* и *в конце*. Т.к. размер набора бесконечен, то поток никогда не заканчивается; кроме того, *Случайное число* не может отвечать на сообщение *содержимое*. Оно может отвечать на сообщения *делать:*, но метод никогда не закончит своё выполнение без специального вмешательства программиста.

Далее представлена реализация класса *Случайное число*; примеры использования экземпляров класса можно посмотреть в главах 11 и 21. Реализация методов *делать:* и *следующий совпадает с: объект* наследуются от класса *Поток*.

имя класса Случайное число
надркласс Поток
имена переменных экземпляра затравка

методы класса

создание экземпляров

новый
↑ сам основной новый присвоить затравку.

методы экземпляра

проверки

в конце
↑ ложь.

доступ

следующий

```
"Lehmer's linear congruential method with modulus m = 2 raisedTo:
16, a = 27181 odd, and 5 = a
8, c = 13849 odd, and c/m approximately 0.21132"
  | врем |
  [
    затравка — 13849 + (27181 * затравка) побитовое и: 8o77777.
    врем ← затравка / 65536.0.
    врем = 0. ]
    пока истина.
  ↑ врем.
```

собственные

присвоить затравку

```
"Чтобы получить псевдо случайное значение затравки, берётся
время часов системы. Это значение является большим положитель-
ным целым, берутся только 16 младших битов."
  затравка ← Время значение часов в миллисекундах побитовое и:
8o77777.
```

Другой пример потока генерируемых элементов это случайное распределение использующиеся в главе 21. надкласс *Случайного распределения* реализован как подкласс *Потока*. Сообщение *следующий: целое* наследуется от *Потока*. Каждый вид *Случайного распределения* определяет является ли он объектом «только для чтения» и, если это так, реализует *пом следующим:* как *сам не должен реализовывать*. Класс *Пространство выбора*, другой пример из главы 21, управляет набором элементов данных и реализует *пом следующим: объект* как добавление к набору.

12.4 Потоки для наборов без внешних ключей

Третим способом хранения потоком ссылки на положение в наборе, упомянутом во введении к этой главе, было хранение нечисловой ссылки. Это полезно в случаях когда элементы набора не могут

быть получены при помощи внешнего ключа или когда этот способ не самый эффективный.

Поток через экземпляры класса *Связанный список* это пример в котором элементы могут быть получены при помощи номеров, но каждый такой доступ требует поиска по цепочке связанных элементов. Более эффективно хранить ссылку на текущий элемент в наборе (вид *Связи*) и затем получать доступ к следующему элементу запрашивая следующую связь у текущего элемента. Такой поток может и читать и писать в *Связанный список*.

Допустим создан подкласс *Потока* который назван *Поток связанного списока*. Каждый экземпляр хранит ссылку на *Связанный список* и ссылку на элемент этого набора. Т.к. поддерживается и чтение и запись, то должны быть реализованы сообщения *следующий*, *пом следующим:*, *в конце* и *содержимое*. (Заметьте что эти четыре сообщения определены в классе *Поток* как *сам ответственность подкласса*.) Новый экземпляр *Потока связанного списка* создаётся при помощи сообщения *на: связанный список*.

имя класса Поток связанного списка
надркласс Поток
имена переменных экземпляра набор текущий узел

методы класса

создание экземпляров

на: связанный список
↑сам основной новый установить на: связанный список.

методы экземпляра

проверки

в конце
↑текущий узел это пусто.

доступ

следующий

```
  | сохр тек узел |
  сохр тек узел ← текущий узел.
 сам в конце ложь: [ текущий узел ← текущий узел следующая связь. ].
  ↑ сохр тек узел.
пом следующим: связь
  | номер пред связь |
  сам в конце истина: [ ↑ набор добавить последним: связь. ].
  номер ← набор номер для: текущий узел.
  номер = 1
    истина: [ набор добавить первым: связь. ]
     ложь: [ пред связь ← набор от: номер − 1. пред связь следующая связь: связь. ].
  связь следующая связь: текущий узел следующая связь.
  текущий узел ← связь следующая связь.
  ↑ связь.
```

собственные

```
установить на: связанный список
  набор ← связанный список.
  текущий узел ← связанный список первый.
```

Чтобы продемонстрировать использование этого нового вида *Потока* создадим *Связанный список* узлов являющихся экземплярами класса *Связь слово*; класс *Связь слово* это подкласс *Связи* который хранит *Цепь* или *Символ*.

имя класса Связь слово
надркласс Связь
имена переменных экземпляра слово

методы класса

создание экземпляров

```
для: цепь
  ↑ сам новый слово: цепь.
```

методы экземпляра

доступ

слово
↑слово.
слово: цепь
слово ← цепь.

сравнение

= связь слово
↑слово = связь слово слово.

печать

печатать в: поток
поток пом следующими все: 'Связь слово для '.
поток пом следующими все: слово.

Из выше упомянутого видно что экземпляр *Связи слова* для слова #один создаётся при помощи:

Связь слово для: #один.

Его цепь для печати это:
'Связь слово для один'

Затем можно создать *Связанный список* из *Связей слов* и *Поток связанного списка* которому доступен это *Связанный список*.

список ← Связанный список новый.
список добавить: (Связь слово для: #один).
список добавить: (Связь слово для: #два).
список добавить: (Связь слово для: #три).
список добавить: (Связь слово для: #четыре).
список добавить: (Связь слово для: #пять).
доступ ← Поток связанного списка на: список.

Пример последовательности сообщений доступу:

предложение	результат

доступ следующий.	Связь слово для один
доступ следующий.	Связь слово для два
доступ следующий совпадает с: (Связь слово для: #три).	истина
доступ пом следующим: (Связь слово для: #вставить).	Связь слово для вставить
доступ содержимое.	Связанный список (Связь слово для один Связь слово для два Связь слово для три Связь слово для вставить Связь слово для пять)
доступ следующий.	Связь слово для пять
доступ в конце.	истина

Аналогично, перебор узлов древовидной структуры, такой как класс "Дерево" данный в одиннадцатой главе, можно осуществить при помощи вида "Потока" который хранит ссылку на текущий узел и затем получает следующий элемент перебирая левое дерево текущего узла, корень или правое дерево. Реализация такого вида "Потока" немного сложнее чем реализация для "Связанного списка" т.к. нужно хранить информацию о том было ли обойдено левое или правое дерево и ссылку на родительский узел текущего узла. Порядок обхода дерева может быть реализован в "Потоке", игнорируя метод которым поддеревья были добавлены в дерево. Так, несмотря на то что мы использовали внутренний порядок обхода в реализации класса "Дерево" и класса "Узел", можно организовать поток через дерево в обратном порядке соответственно реализовав сообщения следующий и пом следующим:.

12.5 Внешние потоки и *Поток файла*

Потоки которые мы рассмотрели только что предполагают что элементы набора могут быть любыми объектами, независящими от представления. Однако для взаимодействия с устройствами ввода вывода, такими как диск, это предположение не верно. В этом слу-

чае элементы хранятся как двоичные байты к которым обычно осуществляется доступ как к числам, цепям, словам (двойным байтам) или байтам. Поэтому требуется поддержка смеси неоднородных сообщений для чтения и записи этих кусков информации разного размера.

Класс *Внешний поток* это подкласс класса *Поток чтения записи.* Его предназначение добавить неоднородный протокол доступа. В него входит протокол для позиционирования и для доступа.

Протокол экземпляров *Внешнего потока*

неоднородный доступ

следующее число: н
Возвращает следующие н байтов набора доступного получателю как положительное *Малое целое* или как *Большое положительное целое.*

следующим числом: н **пом:** з
Помещает аргумент, з, являющийся положительным *Малым целым* или *Большим положительным целым*, в качестве следующих н байтов набора доступного получателю. Если требуется, дополненных нулями.

следующая цепь
Возвращает *Цепь* созданную из следующих элементов набора доступного получателю.

пом следующими цепь: цепь
Помещает аргумент, цепь, в набор доступный получателю.

следующее слово
Возвращает следующие два байта из набора доступного получателю, представленные как *Целое.*

пом следующим словом: целое
Помещает аргумент, целое, в качестве двух следующих байтов набора доступного получателю.

неоднородное позиционирование

заполнить до: размер **пом:** знак

Пропустить, записывая аргумент, знак, в набор доступный получателю столько раз чтобы указатель ссылка дошёл до границы кратной аргументу размер. Возвращает количество записанных знаков.

Класс *Поток файла* это подкласс *Внешнего потока*. Все обращения к внешним файлам происходят при помощи экземпляров *Потока файла*. *Поток файла* работает как будто он имеет доступ к большой последовательности байтов или знаков; предполагается что элементами последовательности являются байты или знаки. Протокол *Потока файла* в точности такой же как у класса *Внешний поток* и его надклассов.

Глава 13

Реализация основного протокола *Набора*

Оглавление

Протокол классов в ирархии *Набора* был описан в главах 9 и 10. В этой главе представлена полная реализация класса *Набор* и реализация основного протокола создания экземпляров, доступа, проверок, добавления, удаления и перебора для каждого подкласса *Набора*. Эти реализации широко используют каркасные сообщения класса *Набор* которые уточняются в этих подклассах. Сообщения *Набора* реализованы в очень общем виде или как *сам ответственность подкласса*. Методы реализуются как *сам ответственность подкласса* если метод зависит от представления экземпляра. Каждый класс должен переопределить такие сообщения

чтобы выполнить все «ответственности подкласса». Подклассы могут переопределять и другие сообщения, из соображений эффективности, на новый метод который использует преимущества представления. Подклассы могут реализовывать некоторые методы как *сам не должен реализовыватъ* что означает что сообщение не должно посылаться экземплярам класса. Например, *Набор последовательностъ* не может отвечать на сообщение *удалитъ:если нету:*; поэтому метод реализован как *сам не должен реализовыватъ*.

13.1 Класс *Набор*

Протокол *Набора* создание экземпляров

В дополнение к сообщениям *новый* и *новый:*, экземпляры *Набора* можно создавать при помощи одного из четырёх сообщений состоящих из одного, двух, трёх или четырёх ключевых слов *с:*. Сообщения *новый* и *новый:* не переопределяются *Набором*; они производят экземпляр являющийся пустым набором. Каждое из четырёх оставшихся методов создания экземпляра определяется *Набором* подобным образом. Сначала создаётся экземпляр (при помощи выражения *сам новый*) и затем к экземпляру, по порядку, добавляются аргументы. Новый экземпляр возвращается в качестве результата. Экземпляр создаётся при помощи *сам новый*, а не при помощи *над новый* или *сам основной новый*, т.к. подкласс *Набора* может переопределить сообщение новый. Любой подкласс *Набора* который представляет объекты фиксированного размера с нумерованными переменными экземпляра должен переопределить следующие сообщения создания экземпляра т.к. такой подкласс не предоставляет реализации для сообщения новый.

имя класса Набор
надркласс Объект

методы класса

создание экземпляра

с: объект

```
  ↑сам новый добавить: объект; себя.
с: первый объект с: второй объект
  ↑сам новый добавить: первый объект; добавить: второй объект;
себя.
с: первый объект с: второй объект с: третий объект
  ↑сам новый
    добавить: первый объект;
    добавить: второй объект;
    добавить: третий объект;
    себя.
с: первый объект с: второй объект с: третий объект с: чет-
вёртый объект
  ↑сам новый
    добавить: первый объект;
    добавить: второй объект;
    добавить: третий объект;
    добавить: четвёртый объект;
    себя.
```

Реализация каждого из сообщений создания экземпляра зависит от возможности вновь созданного экземпляра отвечать на сообщение *добавить:*. Класс *Набор* не может предоставить реализацию для следующих сообщений т.к. они зависят от представления используемого подклассом:

```
добавить: объект
удалить: объект если нету: блок
делать: блок
```

Все остальные сообщения основного протокола набора реализованы в терминах этих трёх сообщений. Каждый подкласс должен реализовать эти три основных сообщения; и затем он может переопределить другие сообщения для улучшения производительности.

Протокол *Набора* добавление

Протокол добавления элементов в набора реализован в классе *Набор* следующим образом.

имя класса Набор

добавление

```
добавить: новый объект
  сам ответственность подкласса.
добавить все: набор
  набор делать: [:каждый | сам добавить: каждый.].
  ↑набор.
```

Заметьте что реализация *добавить все:* зависит и от *делать:* и от *добавить:*. Порядок в котором элемента добавляются из аргумента, *набор*, зависит и от порядка в котором набор перебирает свои элементы (*делать:*) и от способа которым элемент включаются в набор (*добавить:*).

Протокол *Набора* удаление

Сообщение *удалить:* и *удалить все:* реализованы в терминах основного сообщения *удалить:если нету:*, которое должно быть предоставлено подклассом. Эти методы сообщают об ошибке если удаляемый элемент не находится в наборе. Метод *удалить:если нету:* может быть использован для задания различного поведения при исключении.

имя класса Набор

удаление

```
удалить: старый объект если нету: блок исключение
  сам ответственность подкласса.
удалить: старый объект
  ↑сам
    удалить: старый объект
    если нету: [сам ошибка не найден: старый объект.].
удалить все: набор
  набор делать: [:каждый | сам удалить: каждый.].
  ↑набор.
```

собственные

ошибка не найден: объект
  сам ошибка: 'Object is not in the collection.'.

Как обычно, категория собственные указывает на сообщения введённые для поддержки реализации других сообщений; они не используются другими объектами. Большинство сообщений об ошибке которые используются более раза должны быть определены как собственные сообщения чтобы создавать литерал строки сообщения только раз.

Протокол *Набора* проверки

Все сообщения протокола проверок состояния набора могут быть реализованы в *Наборе*.

имя класса Набор

проверки

пустой
  ↑ сам размер = 0.
содержит: объект
  ↑ сам любой удовлетворяет: [**:каждый** | каждый = объект.].
вхождений: объект
  | счёт |
  счёт ← 0.
  сам делать: [**:каждый** | объект = каждый истина: [счёт ← счёт + 1.].].
  ↑ счёт.

Реализация *содержит:* и *вхождений:* зависит от реализации подклассом основного сообщения перебора *делать:*. Аргумент блок *делать:* метода *содержит:* прекращает выполнение сразу как только найден элемент равный аргументу. Если такого элемента не найдено, то выполняется последнее предложение (↑ ложь.). Ответ на сообщения *пустой* и *включает:* это *Логический* объект, *истина* или *ложь*. Сообщение *размер* наследуется от класса *Объект*,

но оно переопределяется в *Наборе* т.к. *размер*, как он определён в *Объекте*, не равен нулю только для объектов переменной длинны.

имя класса Набор

доступ

```
размер
  | счёт |
  счёт ← 0.
  сам делать: [:каждый | счёт ← счёт + 1.].
  ↑ счёт.
```

Этот подход вычисления размера набора малоэффективен поэтому, как мы увидим, этот метод переопределён в большинстве подклассов.

Протокол *Набора* перебор

Реализация всех сообщений которые перебирают элементы набора, за исключением сообщения *делать:*, может быть сделана в классе *Набор*.

имя класса Набор

перебор

```
делать: блок
  сам ответственность подкласса.
собрать: блок
  | новый набор |
  новый набор ← сам разновидность новый.
   сам делать: [:каждый | новый набор добавить: (блок значение: каждый).].
  ↑ новый набор.
выявить: блок
  ↑ сам выявить: блок если ни одного: [сам ошибка не найден: блок.].
выявить: блок если ни одного: блок исключение
```

```
  сам делать: [ :каждый | ( блок значение: каждый ) истина: [ ↑ каждый. ].
пусто. ].
  ↑ блок исключение значение.
ввести: это значение в: бинарный блок
  | следующее значение |
  следующее значение ← это значение.
  сам
    делать: [
      :каждый |
        следующее значение ← бинарный блок значение: следующее значение значение: каждый. ].
  ↑ следующее значение.
отбросить: блок
  ↑ сам выбрать: [ :элемент | ( блок значение: элемент ) == ложь. ].
собрать: блок
  | новый набор |
  новый набор ← сам разновидность новый.
  сам делать: [ :каждый | новый набор добавить: ( блок значение: каждый ). ].
  ↑ новый набор.
```

В методах связанных с *собрать:* и *выбрать:*, сообщение *разновидность* посылается себе. Это сообщение не было указано в девятой главе т.к. оно не является частью внешнего протокола наборов. Оно идёт под категорией собственные чтобы показать его назначение только для внутреннего использования. Сообщение реализовано в классе *Объект* как возвращающее класс получателя.

имя класса Объект

собственные

разновидность

```
  ↑ сам класс.
```

Поэтому выражение

```
сам разновидность новый
```

означает «создать новый экземпляр того же класса что и получатель». Для некоторых наборов, может быть не подходящим создавать «подобный» экземпляр в этом случае; новый набор который надо создать может быть экземпляром другого класса. Такие наборы переопределяют сообщение разновидность. В частности, *Интервал* отвечает что его разновидность это *Ряд* (т.к. невозможно изменять экземпляры *Интервала*); разновидность *Набора отображения* это разновидность отображаемого набора (т.к. *Набор отображение* работает как объект доступа к этому набору).

Если набор нельзя создать просто послав сообщение *новый* классу, то он должен переопределить сообщения *собратъ:* и *выбратъ:*. Т.к. *отбросить:* реализован в терминах *выбратъ:*, то его не нужно переопределять.

Метод *ввести:в:* выполняет аргумент *блок* один раз для каждого элемента получателя. Блоку также передаётся его собственное значение из прошлого выполнения; начальное значение это аргумент *ввести:*. Конечное значение блока возвращается как значение сообщения *ввести:в:*.

Причина введения двух сообщений *выявить:* и *выявить:если ни одного:* похожа на причину введения двух сообщений удаления, *удалить:* и *удалить:если нету:*. В общем случае (*выявить:*) сообщается об ошибке если ни один элемент ни удовлетворяет критерию поиска; программист может избежать этого сообщения об ошибке определив альтернативное исключение (*выявить:если ни одного:*).

Протокол *Набора* преобразование

Протокол для преобразования из любого набора в *Мешок*, *Множество*, *Упорядоченный набор* или *Сортированный набор* реализован прямым способом — создаётся новый экземпляр требуемого набора, затем к нему добавляется каждый элемент получателя. В большинстве случаев, новый экземпляр имеет тот же размер что и оригинальный набор. В случае *Упорядоченного набора*, элементы добавляются в конец последовательности (*добавить последним:*), несмотря на порядок перебора исходного набора.

имя класса Набор

преобразование

как мешок
  ↑Мешок со всеми: сам.
как упорядоченный набор
  ↑сам как: Упорядоченный набор.
как множество
  ↑Множество со всеми: сам.
как сортированный набор
  ↑сам как: Сортированный набор.
как сортированный набор: блок сортировки
  | сортированный набор |
  сортированный набор ← Сортированный набор новый: сам размер.
  сортированный набор сортирующий блок: блок сортировки.
  сортированный набор добавить все: сам.
  ↑сортированный набор.

Протокол *Набора* печать

Реализация сообщений *печатать в:* и *поместить в:* из *Объекта* переопределены в *Наборе*. Наборы печатаются в виде
имя класса (элемент элемент элемент)

Наборы помещают себя как выражения из которых можно создать набор эквивалентной структуры. Это делается в виде:
((имя класса новый))
или
((имя класса новый) добавить: элемент; себя)
или
((имя класса новый) добавить: элемент; добавить: элемент; себя)
с соответствующим количеством каскадированных сообщений для добавления каждого элемента, в зависимости от того есть ли в наборе ни одного элемента, один или более элементов. Сообщение себя возвращает получателя сообщения. Оно используется в каскадированных сообщениях чтобы гарантировать что результат каскадированного сообщения это получатель. Все объекты отвечают на сообщение себя; оно определено в классе *Объект*.

Общие методы для печати и помещения:

имя класса Набор

печать

```
печатать в: поток
  сам печатать имя в: поток.
  сам печатать элементы в: поток.
поместить в: поток
  | ещё не надо |
  поток пом следующими все: '(('.
  поток пом следующими все: сам класс имя.
  поток пом следующими все: ' new)'.
  ещё не надо ← истина.
  сам
    делать: [
      :каждый |
      ещё не надо
        истина: [ещё не надо ← ложь.]
        ложь: [поток пом следующим: $;.].
      поток пом следующими все: ' add: '.
      поток поместить: каждый.].
  ещё не надо ложь: [поток пом следующими все: '; yourself'.].
  поток пом следующим: $).
```

Эти методы используют экземпляры вида *Потока* которые являются объектами доступа к *Цепи*. Метод *печатать в:* устанавливает предел длинны создаваемой строки; длинные наборы могут напечататься как:
класса (элемент элемент ...и т.д. ...)

Формат печати изменён в некоторых подклассах. *Ряды* не печатают своё имя класса; *Интервалы* печатаются с использованием краткой записи при помощи сообщений *до:* и *до:через:* к *Числу*. *Символ* печатает знаки (без # в литеральной форме *Символа*); *Цепь* печатает свои знаки заключёнными в одинарные кавычки.

Сообщение *поместить в:* переопределено в *Наборе ряде* и в нескольких его подклассах т.к. экземпляры создаются при помощи сообщения *новый: целое* вместо просто сообщения *новый*. *Ряды*,

Цепи и *Символы* помещаются в их литеральной форме. *Интервалы* используют краткую запись *до:* и *до:через:*.

13.2 Подклассы *Набора*

Для каждого подкласса *Набора* показаны методы которые реализуют три требуемых сообщения (*добавить:*, *удалить:если нету:* и *делать:*) и сообщения из протоколов добавления, удаления, проверок и перебора которые переопределены. Новый протокол набора для данного подкласса, как это было сделано в девятой главе, не будет даваться в этой главе.

13.2.1 Класс *Мешок*

Мешок представляет неупорядоченный набор в котором элементы могут встречаться более раза. Т.к. элементы *Мешка* неупорядочены, то сообщения *от:* и *от:пом:* переопределены так чтобы сообщать об ошибке.

Экземпляры *Мешка* содержать экземпляр *Словаря* в качестве единственной переменной экземпляра с именем содержимое. Каждый уникальный элемент *Мешка* это ключ *Ассоциации* в содержимом; значение ассоциации это целое представляющее количество вхождений элемента в мешок. Удаление элемента уменьшает счёт; когда счёт становится меньше единицы, то ассоциация удаляется из содержимого. *Мешок* реализует сообщения *новый*, *размер*, *включает:* и *вхождений:*. Новый экземпляр инициализирует свою переменную экземпляра *Словарём*. Переопределение метода *размер* сделано так чтобы суммировать значения всех элементов содержимого. Аргументы сообщений проверки используются как ключи содержимого. В реализации сообщения *содержит:*, ответственность за проверку переложена на содержимое. Чтобы ответить на запрос *вхождений: объект*, метод проверяет что объект содержится в качестве ключа в содержимом и затем смотрит значение (*счёт*) связанное с ним.

имя класса Мешок
надркласс Набор

имена переменных экземпляра содержимое

методы класса

создание экземпляра

```
новый
  ↑сам новый: 4.
новый: количество элементов
  ↑над новый
    присвоить содержимое: (сам класс содержимого новый: количество элементов).
```

методы экземпляра

доступ

```
от: номер
  сам ошибка не ключевой.
от: номер пом: объект
  сам ошибка не ключевой.
размер
  | счёт |
  счёт ← 0.
  содержимое делать: [:каждый | счёт ← счёт + каждый. ].
  ↑счёт.
```

проверки

```
содержит: объект
  ↑содержимое содержит ключ: объект.
вхождений: объект
  (сам содержит: объект) истина: [↑содержимое от: объект. ].
  ↑0.
```

собственные

```
присвоить содержимое: словарь
  содержимое ← словарь.
```

(в Наборе)

ошибка не ключевой
  сам
    ошибка: ('Instances of {1} do not respond to keyed accessing messages.'
      переведённый
        формат: {сам класс имя.}).

Добавление элемента это добавление его один раз, но *Мешок* может добавлять и несколько раз. Реализация *добавить:* вызывает *добавить:с вхождениями:*. Удаление элемента проверяет количество вхождений, уменьшая счёт или удаляя элемент как ключ из содержимого когда счёт становится меньше единицы.

имя класса Мешок

добавление

добавить: новый объект
  ↑сам добавить: новый объект с вхождениями: 1.
добавить: новый объект с вхождениями: целое
  содержимое
    от: новый объект
    пом: (содержимое от: новый объект если нету: [0.]) + целое.
  ↑новый объект.

удаление

удалить: старый объект если нету: блок исключение
  | количество |
  количество ← содержимое от: старый объект если нету: [↑блок исключение значение.].
  количество = 1
    истина: [содержимое удалить ключ: старый объект.]
    ложь: [содержимое от: старый объект пом: количество − 1.].
  ↑старый объект.

Перебор элементов *Мешка* означает выбор каждого элемента *Словаря* и выполнение блока для каждого ключа этого элемента (т.к. в действительности элемент *Мешка* это ключ *Словаря*). Блок выполняется много раз, один раз для каждого вхождения элемента, как указано значением связанным с ключём.

имя класса Мешок

перебор

делать: **блок**

содержимое

ассоциации делать: [**:ассоц** | ассоц значение раз повторить: [блок значение: ассоц ключ.].].

13.2.2 Класс *Множество*

Элементы *Множеств* как и элементы *Мешков* неупорядочены, поэтому сообщения *от:* и *от:пом:* вызывают ошибку. *Множество* не может содержать элемент более одного раза, поэтому, каждое добавление элемента должно, теоретически, проверять весь набор. Чтобы избежать поиска по всем элементам, *Множество* определяет где начать поиск в своих нумерованных переменных при помощи техники хэширования.

У *Множества* есть переменная экземпляра с именем счёт. Счёт содержит количество элементов предотвращая неэффективное определение размера *Множества* при помощи подсчёта не пустых элементов. Поэтому методы *новый*, *новый:* и *размер* переопределены; первые два чтобы инициализировать переменную *счёт*, и последний чтобы просто возвращать значение *счёта*.

имя класса Множество

надркласс Набор

имена переменных экземпляра счёт ряд

методы класса

создание экземпляра

новый
↑сам основной новый инициализировать: 5.
новый: количество элементов
↑над основной новый
инициализировать: (сам размер для: количество элементов).

методы экземпляра

доступ

размер
↑счёт.

Собственное сообщение *Множества*, *найти элемент или пусто:*, хэширует аргумент для получения номера с которого начинается просмотр *Множества*. Просмотр продолжается до тех пор пока не будет найден аргумент, *объект*, или не встретится значение *пусто*. Ответ это номер последнего проверенного элемента. Поэтому сообщения проверки реализованы так:

имя класса Множество

проверки

содержит: объект
↑(ряд от: (сам найти элемент или пусто: объект)) ~~ пусто.
вхождений: объект
↑(сам содержит: объект) истина: [1.] ложь: [0.].

добавление

добавить: новый объект
"Include newObject as one of the receiver's elements, but only if not already present. Answer newObject."
| номер |
новый объект

```
        пусто: [сам ошибка: 'Sets cannot meaningfully contain nil as an
element'.].
    номер ← сам найти элемент или пусто: новый объект.
    (ряд от: номер) пусто: [сам от нового номера: номер пом: новый объект.].
    ↑новый объект.
```

удаление

```
удалить: старый объект если нету: блок
    | номер |
    номер ← сам найти элемент или пусто: старый объект.
    (ряд от: номер) пусто: [↑блок значение.].
    ряд от: номер пом: пусто.
    счёт ← счёт − 1.
    сам устранить конфликт в: номер.
    ↑старый объект.
```

перебор

```
делать: блок
    | каждый |
    счёт = 0 истина: [↑сам.].
    1
        до: ряд размер
        делать: [:номер | (каждый ← ряд от: номер) не пусто: [блок
значение: каждый.].].
```

Сообщение *удалить:если нету:* вызывает метод *найти элемент или пусто:*; если элемент, *старый объект*, не находится, то выполняется аргумент *блок*. Чтобы гарантировать что техника хэширования работает правильно, при удалении одного из элементов оставшиеся требуется уплотнить (*устранить конфликт в:*).

13.2.3 Класс *Словарь*

Словарь это набор *Ассоциаций*. Класс *Словарь* использует технику хэширования для поиска своих элементов которые подобны

элементам его надкласса, *Множества*, но хэшируются ключи *Ассоциаций* вместо самих *Ассоциаций*. Большинство методов доступа *Словаря* переопределены чтобы обращаться к значениям *Ассоциаций* как к элементам, а не к самим *Ассоциациям*.

Словарь реализует сообщения *от:* и *от:пом:*, так что аргумент связанный с ключевым словом *от:* может быть любым ключём *Словаря* (не обязательно *Целым* номером). Аргумент сообщения *содержит:* это значения одной из *Ассоциаций Словаря*, а не одна из *Ассоциаций*. Сообщение *делать:* перебирает значения, а не *Ассоциации*. Аргумент сообщения *удалить:* это тоже значение, но это не подходящий способ удалить элемент из *Словаря* т.к. на элементы ссылаются ключи. Вместо этого нужно использовать *удалить ассоциацию:* или *удалить ключ:*. Поэтому сообщения *удалить:* и *удалить:если нету:* не должны реализовываться для *Словаря*.

Большинство работы в протоколе доступа делается собственными сообщениями, которые либо наследуются от Множества либо подобными сообщениями для поиска ключа.

имя класса Словарь
надркласс Множество

методы экземпляра

доступ

```
от: ключ
  ↑сам от: ключ если нету: [сам ошибка ключ не найден.].
от: ключ пом: объект
  | номер ассоц |
  номер ← сам найти элемент или пусто: ключ.
  ассоц ← ряд от: номер.
  ассоц
    пусто: [
      сам
        от нового номера: номер
        пом: ( Ассоциация ключ: ключ значение: объект ).]
    не пусто: [ассоц значение: объект.].
  ↑объект.
```

```
от: ключ если нету: блок
  | ассоц |
  ассоц ← ряд от: (сам найти элемент или пусто: ключ).
  ассоц пусто: [↑блок значение.].
  ↑ассоц значение.
```

проверки

```
содержит: объект
  сам делать: [:каждый | объект = каждый истина: [↑истина.].].
  ↑ложь.
```

добавление

```
добавить: ассоциация
  | номер элемент |
  номер ← сам найти элемент или пусто: ассоциация ключ.
  элемент ← ряд от: номер.
  элемент
    пусто: [сам от нового номера: номер пом: ассоциация.]
    не пусто: [элемент значение: ассоциация значение.].
  ↑ассоциация.
```

удаление

```
удалить: объект если нету: блок исключение
  сам не должен реализовывать.
```

перебор

```
делать: блок
  над делать: [:ассоц | блок значение: ассоц значение.].
```

собственные

```
ошибка ключ не найден
  сам ошибка: 'key not found'.
```

Заметьте подобие между *от:пом:* и *добавить:*. Разница между ними заключается в действиях совершаемых в случае отсутствия элемента, в случае *от:пом:* создаётся новая *Ассоциация* и запоминается в *Словаре*, в случае *добавить: аргумент*, ассоциация, запоминается так чтобы любые разделяемые ссылки на *Ассоциацию* сохранились.

Сообщение *собрать:* переопределено чтобы избежать проблем со сбором возможно идентичных значений во *Множество* которое будет отбрасывать повторения. Сообщение *выбрать:* переопределено чтобы выбрать *Ассоциации* и применить их значение в качестве аргумента блока.

имя класса Словарь

перебор

```
собрать: блок
  | новый набор |
  новый набор ← Упорядоченный набор новый: сам размер.
  сам делать: [:каждый | новый набор добавить: (блок значение: каждый ). ].
  ↑ новый набор.
выбрать: блок
  | новый набор |
  новый набор ← сам разновидность новый.
  сам
    ассоциации делать: [
      :каждый |
      (блок значение: каждый значение )
        истина: [новый набор добавить: каждый. ]. ].
  ↑ новый набор.
```

Тождественный словарь переопределяет *от:*, *от:пом:* и *добавить:* чтобы сделать проверку ключей на идентичность вместо проверки на равенство.

13.2.4 Наборы последовательности

Набор последовательность это надкласс для всех наборов чьи элементы упорядочены. Из рассмотренных сообщений, селектор *удалить:если нету:* задан как недопустимый для *Наборов последовательностей*, т.к. порядок элементов задаётся внешним образом и предполагается что элементы удаляются в заданном порядке. Т.к. *Набор последовательность* упорядочен, то элементы доступны при помощи сообщения *от:*; реализация предоставляется классом *Объект*. Сообщение *делать:* реализовано путём доступа к каждому элементу от номера 1 до размера набора. *Набор последовательность* создаётся при помощи сообщения *новый:*. Однако, *собрать:* и *выбрать:* должны быть переопределены чтобы создавать новый набор при помощи сообщения *новый:* вместо сообщения *новый*. Методы для селекторов *собрать:* и *выбрать:* показанные ниже используют *Поток записи* для доступа к новому набору, и сообщение *от:* чтобы получить элементы оригинального набора.

имя класса Набор последовательность
надкласс Набор

методы экземпляра

удаление

удалить: старый объект если нету: блок исключение
сам не должен реализовывать.

перебор

делать: блок
1 до: сам размер делать: [**:номер** | блок значение: (сам от: номер).].
собрать: блок
| новый набор |
новый набор ← сам разновидность новый: сам размер.
1
до: сам размер
делать: [

```
        :номер |
      новый набор от: номер пом: ( блок значение: ( сам от: номер ) ). ].
  ↑ новый набор.
выбрать: блок
  | поток |
  поток ← Поток записи на: ( сам разновидность новый: сам размер ).
  1
    до: сам размер
    делать: [
      :номер |
      ( блок значение: ( сам от: номер ) )
        истина: [ поток пом следующим: ( сам от: номер ). ]. ].
  ↑ поток содержимое.
```

13.2.5 Подклассы *Набора последовательности*

Класс *Связанный список*

Элементы *Связанного списка* это экземпляры *Связи* или её подкласса. У каждого *Связанного списка* есть две переменные экземпляра, ссылки на первый и на последний элементы. Добавление элемента понимается как добавление в конец (*добавить последним:*); метод *добавить последним:* делает добавляемый элемент следующим за текущим последним элементом. Удаление элемента предполагает что связь предшествующего элемента должна ссылаться на последующий элемент (или *пусто*). Если удаляемый элемент является первым элементом, то его последующая связь становится первым элементом.

имя класса Связанный список
надкласс Набор последовательность
имена переменных экземпляра первая связь последняя связь

методы экземпляра

доступ

```
от: номер
  | н |
```

```
  н ← 0.
  сам делать: [:связь | ( н ← н + 1 ) = номер истина: [ ↑ связь. ]. ].
  ↑ сам ошибка границы номера: номер.
```

добавление

```
добавить: связь
  ↑ сам добавить последним: связь.
добавить последним: связь
  сам пустой
    истина: [ первая связь ← связь. ]
    ложь: [ последняя связь следующая связь: связь. ].
  последняя связь ← связь.
  ↑ связь.
```

удаление

```
удалить: связь если нету: блок
  | врем связь |
  связь == первая связь
    истина: [
      первая связь ← связь следующая связь.
     связь == последняя связь истина: [ последняя связь ← пусто. ]. ]
    ложь: [
      врем связь ← первая связь.
      [
        врем связь пусто: [ ↑ блок значение. ].
        врем связь следующая связь == связь. ]
          пока ложь: [ врем связь ← врем связь следующая связь. ].
      врем связь следующая связь: связь следующая связь.
    связь == последняя связь истина: [ последняя связь ← врем связь. ]. ].
  связь следующая связь: пусто.
  ↑ связь.
```

перебор

```
делать: блок
  | связь |
```

```
связь ← первая связь.
[связь == пусто.]
 пока ложь: [блок значение: связь. связь ← связь следующая связь.].
```

Пустая связь указывает на конец *Связанного списка*. Поэтому сообщение перебора *делать:* реализовано как простой цикл который продолжается до тех пор пока не встретится *пусто*.

Класс *Интервал*

Интервалы это *Наборы последовательности* чьи элементы вычисляются. Поэтому сообщения для добавления и удаления не поддерживаются. Т.к. элементы не хранятся явно, то весь доступ (*от:*, *размер* и *делать:*) требует вычислений. Каждый метод проверяет нужно ли последний вычисленный элемент увеличить (положительный шаг) или уменьшить (отрицательный шаг) чтобы определить достигнута ли граница (конец).

имя класса Интервал
надркласс Набор последовательность
имена переменных экземпляра начало конец шаг

методы класса

создание экземпляра

от: **начальное целое** **до:** **конечное целое**
↑сам новый присвоить от: начальное целое до: конечное целое через: 1.
от: **начальное целое** **до:** **конечное целое** **через:** **целое шаг**
↑сам новый присвоить от: начальное целое до: конечное целое через: целое шаг.

методы экземпляра

доступ

размер
шаг < 0

```
    истина: [начало < конец истина: [↑ 0. ]. ↑ конец − начало // шаг + 1. ]
    ложь: [конец < начало истина: [↑ 0. ]. ↑ конец − начало // шаг + 1. ].

от: целое
  ( целое >= 1 и: [целое <= сам размер. ] )
    истина: [↑ начало + ( шаг * ( целое − 1 ) ). ].
  сам ошибка границы номера: целое.
от: целое пом: объект
  сам ошибка: 'you can not store into an interval'.
```

добавление

```
добавить: новый объект
  сам не должен реализовывать.
```

удаление

```
удалить: новый объект
  сам ошибка: 'elements cannot be removed from an Interval'.
```

перебор

```
делать: блок
  | значение |
  значение ← начало.
  шаг < 0
    истина: [
      [конец <= значение. ]
      пока истина: [блок значение: значение. значение ← значение + шаг. ].
      пусто. ]
    ложь: [
      [конец >= значение. ]
      пока истина: [блок значение: значение. значение ← значение + шаг. ].
      пусто. ].
собрать: блок
  | след значение результат |
  результат ← сам разновидность новый: сам размер.
  след значение ← начало.
```

```
1
  до: результат размер
  делать: [
    :н |
    результат от: н пом: (блок значение: след значение).
    след значение ← след значение + шаг. ].
↑ результат.
```

собственные

```
присвоить от: начальное целое до: конечное целое через: целое шаг
  начало ← начальное целое.
  конец ← конечное целое.
  шаг ← целое шаг.
```

Наборы ряды — *Ряд*, *Ряд байтов*, *Цепь*, *Текст* и *Символ*

Набор ряд это подкласс *Набора последовательности*; каждый *Набор ряд* это объект переменной длинны. Все методы создания экземпляра переопределены чтобы использовать *новый:*, а не *новый*. Наборы ряды имеют фиксированную длину поэтому сообщение *добавить:* недопустимо; в его надклассе *удалить:* и *делать:* уже запрещены. Поэтому определено только сообщение *размер* как примитив системы который возвращает количество нумерованных переменных экземпляра.

Из подклассов *Набора ряда Ряд* и *Ряд байтов* не переопределяют сообщений которые был рассмотрены в этой главе. Методы доступа для *Цепи* — *от:*, *от:пом:* и *размер* — это примитивы системы; в *Тексте* все сообщения доступа направляются к переменной экземпляра *цепь* (которая является экземпляром *Цепи*). *Символ* не допускает сообщения *от:пом:* и возвращает *Цепь* в качестве своей разновидности.

Упорядоченные наборы и *Сортированные наборы*

Упорядоченный набор хранит порядок, непрерывную последовательность элементов. Т.к. *Упорядоченный набор* расширяем, то

эффективность этого набора получается за счёт выделения дополнительного пространства для последовательности. Две переменных экземпляра, *первый номер* и *последний номер* указывают на первый и последний элемент в последовательности.

Номер в *Упорядоченном наборе* преобразовывается методами доступа (*от:* и *от:пом:*) в диапазон между первым номером и последним номером, и *размер* это просто разность между между этими двумя номерами плюс один. Добавление элемента понимается как добавление в конец; если нету места в конце, то набор копируется с дополнительным выделенным пространством (*создать пространство в конце* это собственное сообщение которое выполняет эту работу). Действительное положение для запоминания элемента вычисляется как положение после последнего номера. Если элемент удаляется, то оставшиеся элемент должны быть сдвинуты чтобы последовательность оставшихся элементов осталась непрерывной (*удалить номер:*).

имя класса Упорядоченный набор
надкласс Набор последовательность
имена переменных экземпляра ряд первый номер последний номер

методы класса

создание экземпляра

```
новый
  ↑сам новый: 10.
новый: целое
  ↑над основной новый присвоить набор: (Ряд новый: целое).
```

методы экземпляра

доступ

```
размер
  ↑последний номер − первый номер + 1.
от: целое
  (целое < 1 или: [целое + первый номер − 1 > последний номер.])
    истина: [сам ошибка нету такого элемента.]
```

```
      ложь: [↑ряд от: целое + первый номер − 1.].
от: целое пом: объект
  | номер |
  номер ← целое как целое.
  (номер < 1 или: [номер + первый номер − 1 > последний номер.])
    истина: [сам ошибка нету такого элемента.]
    ложь: [↑ряд от: номер + первый номер − 1 пом: объект.].
```

добавление

```
добавить: новый объект
  ↑сам добавить последним: новый объект.
добавить последним: новый объект
  последний номер = ряд размер истина: [сам создать пространство
в конце.].
  последний номер ← последний номер + 1.
  ряд от: последний номер пом: новый объект.
  ↑новый объект.
```

удаление

```
удалить: старый объект если нету: блок исключение
  первый номер
    до: последний номер
    делать: [
      :номер |
      старый объект = (ряд от: номер)
        истина: [сам удалить номер: номер. ↑старый объект.].].
  ↑блок исключение значение.
```

собственные

```
ошибка нету такого элемента
  сам ошибка: 'attempt to index non−existent element in an ordered
collection'.
```

Каждое из сообщений перебора *делать:*, *собрать:* и *выбрать:* переопределено — *делать:* чтобы предоставить большую производительность чем метод предоставленный *Набором последовательностью*.

имя класса Упорядоченный набор

перебор

```
делать: блок
  первый номер
    до: последний номер
    делать: [:номер | блок значение: (ряд от: номер).].
собрать: блок
  | новый набор |
  новый набор ← сам разновидность новый: сам размер.
  первый номер
    до: последний номер
    делать: [
      :номер |
       новый набор добавить последним: (блок значение: (ряд от:
номер)).].
  ↑новый набор.
выбрать: блок
  | новый набор элемент |
  новый набор ← сам пустая копия.
  первый номер
    до: последний номер
    делать: [
      :номер |
      (блок значение: (элемент ← ряд от: номер))
        истина: [новый набор добавить последним: элемент.].].
  ↑новый набор.
```

В методе *выбрать:* новый набор создаётся при помощи посылки сообщения *пустая копия* оригинальному набору. Это сообщение создаёт новый набор с достаточным выделенным пространством чтобы поместить все элементы оригинала, однако не все элементы могут

быть помещены в этот набор. Таким образом избегаются затраты времени на расширение нового набора.

Сортированный набор это подкласс *Упорядоченного набора*. Сообщение *от:пом:* вызывает ошибку, требуя программиста использовать сообщение *добавитъ:*; *добавитъ:* вставляет новый элемент в соответствии со значением переменной экземпляра *сортирующий блок*. Определение положения вставки производится методом «пузырька». Сообщение *собратъ:* тоже переопределяется чтобы создавать для сбора значений блока *Упорядоченный набор* вместо *Сортированного набора*.

13.2.6 Класс *Набор отображение*

Экземпляры *Набора отображения* имеют две переменные экземпляра — *областъ* и *карта*. Значение области это любой *Словарь* или *Набор последовательностъ*; доступ к его элементам осуществляется через карту. Сообщение *добавитъ:* недопустимо. Оба сообщения *от:* и *от:пом:* переопределены в *Наборе последовательности* чтобы поддерживать не прямой доступ через карту к элементам области.

имя класса Набор отображение
надкласс Набор последовательность
имена переменных экземпляра область карта

методы класса

создание экземпляра

набор: набор карта: набор последовательность
↑сам основной новый присвоить набор: набор карту: набор последовательность.
новый
сам
ошибка: 'MappedCollections must be created using the collection:map: message'.

методы экземпляра

доступ

```
от: номер
  ↑область от: (карта от: номер).
от: номер пом: объект
  ↑область от: (карта от: номер) пом: объект.
размер
  ↑карта размер.
```

добавление

```
добавить: новый объект
  сам не должен реализовывать.
```

перебор

```
делать: блок
  карта
    делать: [:значение карты | блок значение: (область от: значение карты).].
собрать: блок
  | поток |
  поток ← Поток записи на: (сам разновидность новый: сам размер).
  сам
    делать: [
      :значение области |
      поток пом следующим: (блок значение: значение области).].
  ↑поток содержимое.
выбрать: блок
  | поток |
  поток ← Поток записи на: (сам разновидность новый: сам размер).
  сам
    делать: [
      :значение области |
      (блок значение: значение области)
        истина: [поток пом следующим: значение области.].].
  ↑поток содержимое.
```

собственные

присвоить набор: **набор** **карту:** **словарь**
область ⟵ набор.
карта ⟵ словарь.
разновидность
↑ область разновидность.

Глава 14

Классы поддержки ядра

Оглавление

14.1 Класс *Неопределённый объект*

Объект *пусто* является значением не инициализированных переменных. Также он представляет бессмысленный результат. Он является единственным экземпляром класса *Неопределённый объект*.

Смысл включения в систему класса *Неопределённый объект* заключается в обработке ошибочных сообщений. Типичной ошибкой при выполнении предложения Смолтока является посылка объекту сообщения которое он не понимает. Часто это происходит из за того что переменная не инициализирована правильно — часто переменная которая должна ссылаться на некоторый другой объект ссылается вместо этого на *пусто*. Сообщение об ошибке часто имеет вид:

Объект

- Величина
  - Знак
  - Дата
  - Время
  - Число
    - Плавающее
    - Дробь
    - Целое
      - Большое положительное целое
      - Большое отрицательное целое
      - Малое целое
  - Ключ поиска
    - Ассоциация
- Связь
  - Процесс
- Набор
  - Набор последовательность
    - Связанный список
      - Семафор
    - Набор ряд
      - Ряд
        - Растровое изображение
        - Ряд серий
        - Цепь
          - Символ
        - Текст
        - Ряд байтов
    - Интервал
    - Упорядоченный набор
      - Сортированный набор
  - Мешок
  - Набор отображение
  - Множество
    - Словарь
      - Тождественный словарь
- Поток
  - Позиционируемый поток
    - Поток чтения
    - Поток записи
      - Поток чтения записи
        - Поток файл
  - Случайное число
- Неопределённый объект
- Логика
  - Истина
  - Ложь
- Планировщик исполнителя
- Задержка
- Разделяемая очередь
- Поведение
  - Описание класса
    - Класс
    - Метакласс
- Точка
- Прямоугольник

Имя класса не понимает *селектора сообщения*
Имя класса указывает класс получателя и *селектор сообщения* это селектор неправильно посланного сообщения.

Заметьте что если бы пусто было экземпляром *Объекта*, то сообщение посланное ему вызвало бы сообщение
Объект не понимает *селектора сообщения*
менее ясно чем сказать что неопределённый объект не понимает сообщения. Ценой описания класса получается возможность улучшения сообщения об ошибке.

Проверка на то является ли объект *пусто* осуществляется в классе *Объект*, но переопределена в *Неопределённом объекте*. В классе *Объект*, сообщение *это пусто* и *не пусто* реализованы как:

```
имя класса Объект

это пусто
  ↑ложь.
не пусто
  ↑истина.
```

В классе *Неопределённый объект* сообщения *это пусто* и *не пусто* реализованы как:

```
имя класса Неопределённый объект

это пусто
  ↑истина.
не пусто
  ↑ложь.
```

Поэтому в *Объекте* не требуется проверка условия.

14.2 Классы *Логика*, *Истина* и *Ложь*

Протокол для логических значений предоставляется классом *Логика*; логические значения представлены подклассами *Логики* — *Истиной* и *Ложью*. Подклассы не добавляют нового протокола;

они переопределяют многие сообщения для увеличения производительности методов надкласса. Идея подобна той что используется для проверки на эквивалентность *пусто* в *Объекте* и в *Неопределённом объекте*; *истина* знает что она представляет логическую истину и *ложь* знает что она представляет логическую ложь. Покажем реализацию некоторых методов протокола управления иллюстрирующую эту идею.

Логические операции это:

Протокол экземпляров *Логики*

логические операции

& логическое
Вычисляет конъюнкцию. Возвращает *истину* если и получатель и аргумент это *истина*.

| логическое
Вычисляет дизъюнкцию. Возвращает *истину* если либо получатель либо аргумент это *истина*.

не
Отрицание. Возвращает *истину* если получатель это *ложь*, возвращает *ложь* если получатель это *истина*.

экв: логическое
Возвращает *истину* если получатель эквивалентен аргументу, логическое.

и или: логическое
Исключающее или. Возвращает *истину* если получатель не эквивалентен аргументу, логическое.

Эти операции конъюнкции и дизъюнкции «вычисляются» — это означает что аргумент вычисляется не смотря на значение получателя. Это поведение пртивоположно поведению сообщений *и:* и *или:* в случае которых получатель определяет нужно ли вычислять аргумент.

Протокол экземпляров *Логики*

и: блок альтернатива

Невычисляемая конъюнкция. Если получатель это *истина*, возвращается значение аргумента; иначе, возвращается *ложь* без вычисления аргумента.

или: **блок альтернатива**
Невычисляемая дизъюнкцию. Если получатель это *ложь*, возвращается значение аргумента; иначе возвращается *истина* без вычисления аргумента.

истина: **блок истина** **ложь:** **блок ложь**
Условное выражение. Если получатель это *истина*, возвращается результат выполнения блока истины; иначе возвращается результат выполнения блока лжи.

ложь: **блок ложь** **истина:** **блок истина**
Условное выражение. Если получатель это *истина*, возвращается результат выполнения блока истины; иначе возвращается результат выполнения блока лжи.

истина: **блок истина**
Условное выражение. Если получатель это *истина*, возвращается результат выполнения блока истины; иначе возвращается *пусто*.

ложь: **блок ложь**
Условное выражение. Если получатель это *ложь*, возвращается результат выполнения блока лжи; иначе возвращается *пусто*.

Аргументы сообщений *и:* и *или:* должный быть блоками чтобы отложить вычисления. Условные выражения представлены как сообщения *истина:ложь:*, *ложь:истина:*, *истина:* и *ложь:*, как уже было указано и проиллюстрировано в предыдущих главах. Сообщения реализованы в подклассах *Логики* чтобы выполнился соответствующий блок.

В классе *Истина*, эти методы выглядят так:

имя класса Истина

истина: **блок истина** **ложь:** **блок ложь**
↑блок истина значение.

ложь: **блок ложь** **истина:** **блок истина**
↑блок истина значение.

истина: **блок альтернатива**
↑блок альтернатива значение.

```
ложь: блок альтернатива
  ↑пусто.
```

В классе *Ложь*, эти методы выглядят так:

```
имя класса Ложь
```

```
истина: блок истина ложь: блок ложь
  ↑блок ложь значение.
ложь: блок ложь истина: блок истина
  ↑блок ложь значение.
истина: блок альтернатива
  ↑пусто.
ложь: блок альтернатива
  ↑блок альтернатива значение.
```

Если икс это 3, то

```
икс > 0 истина: [икс ← икс − 1.] ложь: [икс ← икс + 1.].
```

это интерпретируется так: *икс > 0* вычисляется в *истину*, единственный экземпляр класса *Истина*; метод с селектором *истина:ложь:* находится в классе *Истина*, поэтому блок [икс ← икс − 1.] вычисляется без дальнейших проверок.

Таким образом механизм поиска сообщения предоставляет эффективную реализацию условных управляющих структур без добавления дополнительной примитивной операции или циклического определения.

14.3 Дополнительный протокол класса *Объект*

Протокол класса *Объект*, разделяемый всеми объектами, был введён в шестой главе. Несколько категорий сообщений не были включены в это начальное рассмотрение. Большинство сообщений из этой части протокола *Объекта* предоставляют поддержку системе для обработки сообщений, взаимоотношений зависимости, базовой обработки сообщений и примитивов системы.

14.3.1 Взаимоотношения зависимости между объектами

В системе Смолток информация представляется в виде объектов. Переменные объектов ссылаются на объекты; в этом случае объекты явным образом зависят один от другого. Классы связаны с их надклассами и метаклассами; эти классы разделяют внешнее и внутреннее описание и таким образом зависят один от другого. Этот вид зависимости является центральным в семантике языка Смолток. Он согласовывает описывающую информацию между объектами.

Классом *Объект* поддерживается дополнительный вид зависимости. Он предназначен для координации действий с различными объектами. В частности, его назначение заключается в возможности одним объектом, скажем А, ссылаться на другой объект, скажем Б, так чтобы Б был проинформирован об изменении произошедшем каким-либо образом в А. Информирование происходит об изменении А и о характере изменения, Б может решить произвести некоторые действия такие как обновление своего состояния. Поэтому концепция изменений и обновлений это составная часть поддержки этого третьего вида зависимости объектов.

Протокол класса *Объект*:

Протокол экземпляров *Объекта*

доступ к зависимостям

добавить зависимость: **объект**
Добавляет аргумент, объект, в качестве одной из зависимостей получателя.

удалить зависимость: **объект**
Удаляет аргумент, объект, из зависимостей получателя.

зависимости
Возвращает *Упорядоченный набор* объектов которые зависят от получателя, то есть такие которые должны быть уведомлены при изменении получателя.

освободить

Удаляет ссылки на объект которые могут ссылаться обратно на получателя. Это сообщение переопределяется любым подклассом который создаёт ссылки на зависимости; выражение над освободить включается в каждое такое переопределение.

изменение и обновление

изменён
Получатель изменён каким то способом; информирует все зависимости посылкой сообщения *обновить:*.

изменён: параметр
Получатель изменён; изменение описывается параметром, параметр. Обычно аргумент это *Символ* являющийся частью протокола изменения зависимостей; в поведении по умолчанию используется получатель в качестве аргумента. Информируются все зависимости.

обновить: параметр
Объект от которого зависит получатель был изменён. Получатель обновляет своё состояние (по умолчанию ничего не делается).

Рассмотрим в качестве примера объекты которые моделируют светофор. Обычный светофор это объект с тремя лампами, все разного цвета. Только одна из них может быть включена в данный момент. В этом смысле включён или выключен огонь зависит от состояния других двух огней. Есть различные способы создать такую зависимость. Допустим мы создали класс *Лампа* как описано ниже.

имя класса Лампа
надркласс Объект
имена переменных экземпляра состояние

методы класса

создание экземпляра

включенный
↑сам новый включенный.

выключенный
↑сам новый выключенный.

методы экземпляра

состояние

выключить
  сам включен истина: [состояние ⟵ ложь.].
включить
  сам выключен истина: [состояние ⟵ истина. сам изменён.].

проверки

включен
  ↑ состояние.
выключен
  ↑ состояние не.

изменение и обновление

обновить: лампа
  лампа == сам ложь: [сам выключить.].

собственные

включенный
  состояние ⟵ истина.
выключенный
  состояние ⟵ ложь.

Модель очень проста. *Лампа* либо включена либо выключена, поэтому флаг состояния хранится в переменной экземпляра; он равен истине если *Лампа* включена, либо лжи если *Лампа* выключена. Когда *Лампа* включается (*включить*), то она посылает себе сообщение *изменён*. Другие изменения состояния не рассылаются зависимостям т.к. предполагается что *Лампа* выключается в ответ на включение другой *Лампы*. По умолчанию в ответ на сообщение *изменён* всем зависимостям посылается сообщение *обновить:*

сам (т.е. объект который был изменён это аргумент сообщения *обновить:*). Поэтому *обновить:* реализовано в *Лампе* так чтобы выключить её. Если параметр это получатель то, естественно, сообщение игнорируется.

Класс *Светофор* определён так чтобы можно было использовать любое количество связанных ламп. Сообщение создания экземпляра *с:* в качестве аргумента получает число *Ламп*. Каждая *Лампа* зависит от всех остальных *Ламп*. Когда *Светофор* разрушается, зависимости между его *Лампами* устраняются (сообщение наследуемое от класса *Объект* для устранения зависимостей это *освободить*; оно реализовано в *Светофоре* чтобы разослать сообщение всем *Лампам*).

имя класса Светофор
надкласс Объект
имена переменных экземпляра лампы

методы класса

создание экземпляра

с: количество ламп
  ↑сам новый ламп: количество ламп.

методы экземпляра

работа

включить: номер лампы
  (лампы от: номер лампы) включить.

инициализировать освободить

освободить
  над освободить.
  лампы делать: [**:каждая лампа** | каждая лампа освободить.].
  лампы ← пусто.

собственные

```
ламп: количество ламп
  лампы ← Ряд новый: (количество ламп макс: 1).
  лампы от: 1 пом: Лампа включенный.
  2
    до: количество ламп
    делать: [:номер | лампы от: номер пом: Лампа выключенный.].
  лампы
    делать: [
      :каждая лампа |
      лампы
        делать: [
          :зависимая лампа |
          каждая лампа ~~ зависимая лампа
            истина: [каждая лампа добавить зависимость: зависимая лампа.].].].
```

Собственное сообщение инициализации это *ламп: количество ламп*. Все лампы кроме первой создаются выключенными. Затем каждая лампа присоединяется ко всем остальным лампам (используя сообщение *добавить зависимость:*). Моделируемый *Светофор* управляется циклическим алгоритмом, возможно с задержками, последовательно включающим каждую лампу. Простой пример показанный ниже создаёт *Светофор* с включённой первой лампой, и затем включает каждый раз следующую лампу. Моделирование перекрёстка может включать различные варианты управления лампами.

```
светофор ← Светофор с: 3.
светофор включить: 2.
светофор включить: 3.
```

Сообщение *включить:* посланное *Светофору* пересылает сообщение *включить* указанной *Лампе*. Если лампа в этот момент выключена, то она включается и посылается сообщение *изменён*. Сообщение *изменён* посылает сообщение *обновить:* каждой зависимости *Лампы*; если зависимая лампа включена, то она выключается.

Очень важное использование этого протокола используется для поддержания различных графических изображений объекта. Каждое изображение зависит от объекта который она изображает, если объект изменяется, изображение должно быть проинформировано об этом чтобы решить влияет ли изменение на показанную информацию. Интерфейс с пользователем системы Смолток широко использует этот протокол для рассылки уведомлений об изменении объектов; он используется для управления набором возможных действий в меню которые пользователь может выполнить в соответствии с показанной на мониторе информацией. Сами меню могут быть созданы с помощью связывания вместе возможных действий, подобно тому как мы связали вместе лампы светофора.

14.3.2 Обработка сообщений

Все действия в системе Смолток выполняются при помощи посылки сообщений объектам. Из соображений эффективности, экземпляры класса *Сообщение* создаются только при возникновении ошибки когда нужно сохранить состояние сообщения в доступной структуре. Поэтому большинство сообщений в системе не существуют в форме экземпляра *Сообщения* и не передаются объекту.

В некоторых случаях удобно вычислять селектор передаваемого сообщения. Например, допустим что список возможных селекторов сообщения содержатся в объекте и, на основании вычисления, выбирается один из этих селекторов. Допустим он присваивается в качестве значения переменной *селектор*. Затем нужно передать сообщение некоторому объекту, называемому получатель. Нельзя просто написать выражение *получатель селектор* т.к. оно означает посылку объекту на который ссылается переменная *получатель* унарного сообщения *селектор*. Однако можно написать:

получатель выполнить: селектор.

В результате значение аргумента, *селектор*, будет послано в качестве сообщения получателю. Протокол поддерживающий возможность посылать вычисленное сообщение объекту предоставляется классом *Объект*. Этот протокол включает методы для передачи как вычисленных ключевых слов так и унарных сообщений.

Протокол экземпляров *Объекта*

обработка сообщений

выполнить: **символ**
Посылает получателю унарное сообщение указываемое аргументом, символ. Аргумент это селектор сообщения. Сообщается об ошибке если количество аргументов ожидаемых селектором не равно нулю.

выполнить: **символ** **с:** **объект**
Посылает получателю сообщение с ключевыми словами указываемыми аргументами. Первый аргумент, символ, это селектор сообщения. Другой аргумент, объект, это аргумент сообщения которое будет послано. Сообщается об ошибке если количество аргументов ожидаемых селектором не равно одному.

выполнить: **символ** **с:** **первый объект** **с:** **второй объект**
Посылает получателю сообщение с ключевыми словами указываемыми аргументами. Первый аргумент, символ, это селектор сообщения. Другие аргументы, первый объект и второй объект, это аргументы сообщения которое будет послано. Сообщается об ошибке если количество аргументов ожидаемых селектором не равно двум.

выполнить: **символ** **с:** **первый объект** **с:** **второй объект** **с:** **третий объект**
Посылает получателю сообщение с ключевыми словами указываемыми аргументами. Первый аргумент, символ, это селектор сообщения. Другие аргументы: первый объект, второй объект, и третий объект — это аргументы сообщения которое будет послано. Сообщается об ошибке если количество аргументов ожидаемых селектором не равно трём.

выполнить: **символ** **с аргументами:** **ряд**
Посылает получателю сообщение с ключевыми словами указываемыми аргументами. Аргумент, символ, это селектор сообщения. Аргументы сообщения это элементы ряда. Сообщается об ошибке если количество аргументов ожидаемых селектором не равно размеру ряда.

Один из способов использования этого протокола это декодер

команд пользователя. Допустим, например, что нужно смоделировать очень простой калькулятор в котором операнды предшествуют операторам. Возможная реализация представляет калькулятор у которого есть (1) текущий результат, который также является первым операндом, и (2) возможно неопределённый второй операнд. Каждый оператор это селектор сообщения понимаемого результатом. Посылка сообщения *очистить*, в первый раз, сбрасывает операнд; посылка сообщения очистить когда операнд сброшен сбрасывает результат.

имя класса Калькулятор
надкласс Объект
имена переменных экземпляра результат операнд

методы класса

создание экземпляра

новый
↑ над новый инициализировать.

методы экземпляра

доступ

результат
↑ результат.

вычисление

применить: оператор
(сам отвечает на: оператор) ложь: [сам ошибка: 'operation not understood'.].
операнд это пусто
истина: [результат ← результат выполнить: оператор.]
ложь: [результат ← результат выполнить: оператор с: операнд.].
очистить
операнд это пусто истина: [результат ← 0.] ложь: [операнд ← пусто.].

операнд: число
операнд ← число.

собственные

инициализировать
результат ← 0.

Пример показывающий использование класса "Калькулятор".

hp ← Калькулятор новый.

Создаёт *hp* как *Калькулятор*. Переменные экземпляра инициализируются: результат нулём, операнд *пусто*.

hp операнд: 3.

Представте что пользователь нажал кнопку с надписью 3 и задал операнд.

hp применить: #'+'.

Пользователь выбрал сложение. Метод *применить:* определят что оператор понимается и что операнд не пуст; поэтому результату присваивается значение выражения:

результат выполнить: оператор с: операнд.

что эквивалентно

0 + 3

Метод присваивает результату 3; операнд остаётся равным 3 поэтому:

hp применить: #'+'.

опять добавит 3, поэтому сейчас результат это 6.

```
hp операнд: 1.
hp применить: #'-'.
hp очистить.
hp применить: #в квадрате.
```

Результат был равен 6, вычли 1, и вычислили квадрат; сейчас результат равен 25.

14.3.3 Сообщения примитивы системы

Есть несколько сообщений определённых в классе *Объект* чьё назначение поддержка реализации нужд всей системы. Их категория это примитивы системы. Эти методы предоставляют прямой доступ к состоянию экземпляров и, некоторым образом нарушают принцип по которому каждый объект имеет полный контроль над хранимыми в его переменных значениях. Однако доступ к этим переменным нужен интерпретатору языка. Предоставление доступа полезно для создания инструментов среды программирования. Примеры этих сообщений: *пер экз от: целое* и *пер экз от: целое пом: объект* которые соответственно получают и присваивают значения именованных переменных экземпляра.

Протокол экземпляров *Объекта*

примитивы системы

становится: **другой объект**
Обменивает указатели на экземпляры получателя и аргумента, другой объект. Все переменные во всей системе которые указывают на получателя будут после этого указывать на аргумент и наоборот. Сообщается об ошибке если один из объектов это *Малое целое*.

пер экз от: **номер**
Возвращает именованую переменную получателя. Нумерация переменных соответствует порядку в котором переменные определены.

пер экз от: **номер** **пом:** **значение**
Помещает аргумент, значение, в именованую переменную получателя. Нумерация переменных соответствует порядку в котором переменные определены. Возвращает значение.

следующий экземпляр
Возвращает следующий экземпляр после получателя в перечислении всех экземпляров данного класса. Возвращает пусто если были перебраны все экземпляры.

Возможно самый необычный и эффективный примитив системы это сообщение становится: другой объект. В ответ на это сообщение

обмениваются указатели на получателя и аргумент, другой объект. Пример использования этого метода можно найти в реализации сообщения расти нескольких классов наборов. Сообщение расти посылается когда количество элементов которые могут быть сохранены в наборе (фиксированной длинны) нужно увеличить без копирования набора; копирование недопустимо т.к. все разделяемые ссылки на набор должны быть сохранены. Поэтому создаётся новый набор, его элементы присваиваются и затем исходный набор преобразуется (становится) в новый набор. Все указатели на исходный набор заменяются указателями на новый набор.

Следующий пример это метод расти из класса *Словарь методов*.

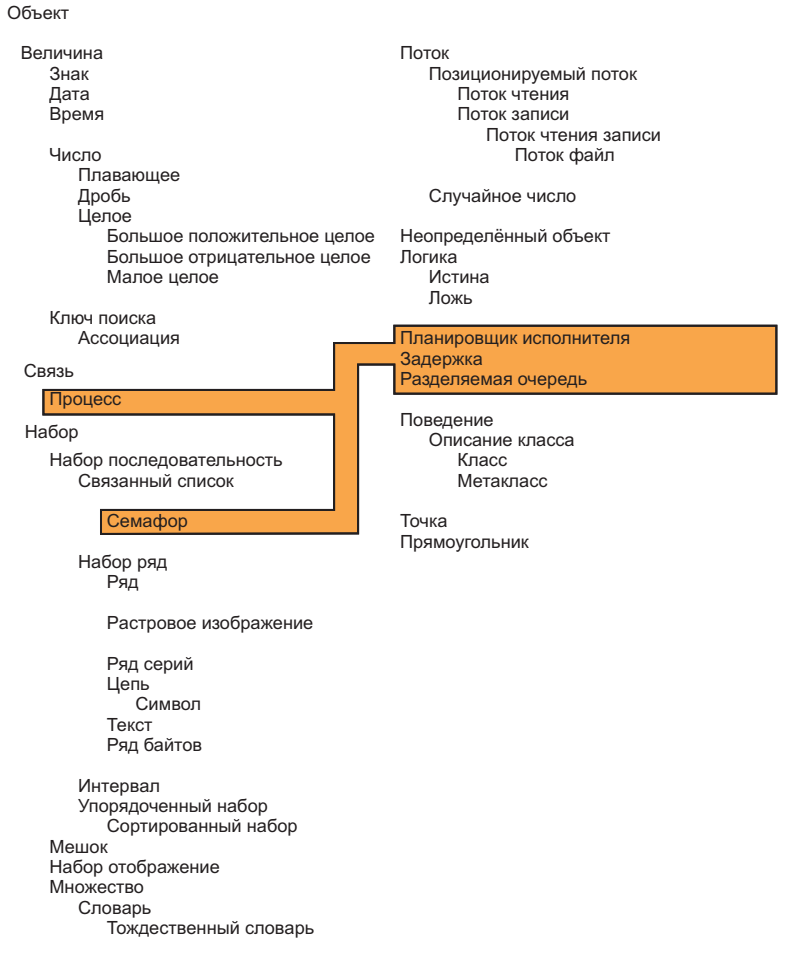
```
расти
  | новый сам ключ |
  новый сам ← сам разновидность новый: сам основной размер.
  1
    до: сам основной размер
    делать: [
      :н |
      ключ ← сам основной от: н.
      ключ не пусто: [новый сам от: ключ пом: (ряд от: н).].].
  сам становится: новый сам.
```

Глава 15

Многочисленные независимые процессы

Оглавление

Система Смолток предоставляет поддержку многочисленных независимых процессов при помощи трёх классов: *Процесса*, *Планировщика исполнителя* и *Семафора*. *Процесс* представляет последовательность действий которые могут быть выполнены независимо от действий представляемых другими *Процессами*. *Планировщик исполнителя* планирует использование виртуальной машины Смолтока которая выполняет действия представляемые в системе *Процессом*. Может существовать много *Процессов* чьи действия выполняются и *Планировщик исполнителя* определят какая из них будет выполняться виртуальной машиной в данный момент. *Сема-*


Объект
Величина
Знак
Дата
Время
Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое
Ключ поиска
Ассоциация
Связь
Процесс
Набор
Набор последовательность
Связанный список
Семафор
Набор ряд
Ряд
Растровое изображение
Ряд серий
Цепь
Символ
Текст
Ряд байтов
Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь
Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл
Случайное число
Неопределённый объект
Логика
Истина
Ложь
Планировщик исполнителя
Задержка
Разделяемая очередь
Поведение
Описание класса
Класс
Метакласс
Точка
Прямоугольник

фор позволяет различным независимым процессам синхронизировать между собой их действия. *Семафоры* предоставляют простую форму синхронного средства связи которая может быть использована для создания более сложных синхронных взаимодействий. Также *Семафоры* предоставляют синхронное средство связи с асинхронными устройствами такими как устройства ввода пользователя и часы реального времени.

Семафоры часто являются не самым удобным механизмом синхронизации. Экземпляры *Разделяемой очереди* и *Задержки*, удовлетворяющие две наиболее частые потребности синхронизации, используют *Семафоры*. *Разделяемая очередь* предоставляет безопасный способ передачи объектов между независимыми процессами, а *Задержка* позволяет процессам синхронизироваться с часами реального времени.

15.1 Процессы

Процесс это последовательность действий описываемая предложениями и выполняемая виртуальной машиной Смолтока. Несколько процессов системы следят за асинхронными устройствами. Например, существует процесс следящий за клавиатурой, за указывающим устройством и за часами реального времени. Также существует процесс следящий за доступной памятью системы. Наиболее важным для пользователя процессом является тот процесс который выполняет действия заданные пользователем, например, редактирование текста, графики или определение класса. Эти процессы интерфейса с пользователем должны поддерживать связь с процессами следящими за клавиатурой и указывающим устройством чтобы определять что делает пользователь.

Новый процесс можно создать послав унарное сообщение *разветвить* блоку. Например, следующее предложение создаёт новый процесс который показывает на мониторе три времени называемых *Восточное время*, *Горное время* и *Тихоокеанское время*.

```
[
    Восточное время показать.
    Горное время показать.
```

```
    Тихоокеанское время показать. ]
      разветвить.
```

Действия производимые новым процессом описываются предложениями блока. Сообщение *разветвить* имеет тот же эффект что и сообщение *значение*, но оно отличается тем как возвращается результат. Когда блок получает сообщение *значение*, он ждёт пока выполнятся все предложения. Например, следующие предложение не вернёт значения до тех пор пока все три времени не покажутся на мониторе.

```
[
  Восточное время показать.
  Горное время показать.
  Тихоокеанское время показать. ]
    значение.
```

Значение возвращаемое из блока при посылке ему сообщения *значение* это значение последнего предложения блока. Когда блок принимает сообщение *разветвить*, управление возвращается сразу, обычно до того как будут выполнены предложения. Это позволяет выполнится предложениям следующим за сообщением *разветвить* независимо от предложений блока. Например, результатом следующих предложений будет сортировка содержимого набора *список имён* независимо от показа трёх времён.

```
[
  Восточное время показать.
  Горное время показать.
  Тихоокеанское время показать. ]
    разветвить.
алфавитный список ← список имён сортированный.
```

Набор может быть отсортирован до того как будут показаны любые из часов либо все часы могут быть показаны до завершения сортировки набора. Может произойти одно из вышеуказанных двух предельных событий или промежуточное событие при котором сортировка и показ часов перемешаны что зависит от способа которым написаны сортировка и показ часов. Два процесса, один из которых посылает сообщения *разветвить* и *сортированный* и тот кото-

рый посылает сообщения *показать*, выполняются независимо. Т.к. предложения блока могут ещё выполняться при возвращении из метода *разветвить*, то значение возвращаемое этим методом должно быть независимо от значений предложений блока. Блок возвращает себя в качестве значения сообщения *разветвить*.

Каждый процесс системы представляется экземпляром класса *Процесс*. В ответ на сообщение *разветвить* блок создаёт новый экземпляр *Процесса* и планирует выполнение своих предложений исполнителем. Блоки также отвечают на сообщение *новый процесс* созданием и возвращением нового экземпляра *Процесса*, но при этом не планируется выполнение его предложений виртуальной машиной. Это сообщение полезно т.к., в отличии от *разветвить*, оно возвращает ссылку на *Процесс*. Процесс созданный сообщением *новый процесс* называется приостановленным т.к. его предложения не выполняются. Например, следующие предложения создают два новых *Процесса* но в результате не будет показано часов и не будет происходить сортировка.

```
процесс показа часов ← [
  Восточное время показать.
  Горное время показать.
  Тихоокеанское время показать. ]
    новый процесс.
процесс сортировки ← [ алфавитный список ← список имён сортированный. ] новый процесс.
```

Действия представляемые одним из этих приостановленных *Процессов* могут быть выполнены при помощи посылки *Процессу* сообщения *возобновить*. Результатом следующих двух предложений должна стать посылка сообщения *показать Восточному времени* и сообщения *сортированный списку имён*.

```
процесс показа часов возобновить.
процесс сортировки возобновить.
```

Т.к. сообщения *показать* и *сортированный* будут посылаться из разных *Процессов*, то их выполнение может быть перемешано. Другим примером использования сообщения *возобновить* является реализация сообщения *разветвить Блока замыкания*.

разветвить

```
↑сам новый процесс возобновить.
```

Дополняющее сообщение, *приостановить*, возвращает *Процесс* в состоянии приостановки в котором исполнитель не выполняет его предложения. Сообщение *завершить* не допускает дальнейшее выполнение *Процесса*, не смотря на то приостановлен он или нет.

Протокол экземпляров *Процесса*

изменение состояния процесса

возобновить
Позволяет продвижение выполнения получателя.

приостановить
Останавливает продвижение выполнения получателя таким образом что выполнение можно позднее возобновить (послав сообщение возобновить).

завершить
Навсегда останавливает продвижение выполнения получателя.

Блоки также понимают сообщение с селектором *новый процесс с:* которое создаёт и возвращает новый *Процесс* передавая значения аргументов блока. Аргумент сообщения *новый процесс с:* это *Ряд* чьи элементы используются как значения аргументов блока. Размер *Ряда* должен быть равен количеству аргументов блока. Например:

```
процесс показа часов ← [ :часы | часы показать. ] новый процесс с:
( Ряд с: Горное время ).
```

Ниже приведён протокол *Контекста блока* который позволяет создавать новые *Процессы*.

Протокол экземпляров *Контекста блока*

планирование

разветвить
Создаёт новый *Процесс* и планирует выполнение предложений получателя.

новый процесс
Возвращает приостановленный *Процесс* выполнения предложений получателя. Новый *Процесс* не планируется.

новый процесс с: аргумент ряд
Возвращает новый приостановленный *Процесс* выполнения предложений получателя со значениями аргументов блока из аргумента ряда.

15.1.1 Планирование

Виртуальная машина Смолтока имеет только один исполнитель способный выполнять одну последовательность действия представляемых *Процессом*. Поэтому когда *Процесс* получает сообщение *возобновить*, то его действия могут не начать выполняться немедленно. *Процесс* чьи действия в текущий момент выполняются называется активным. При получении активным *Процессом* сообщения *приостановить* или *завершить*, выбирается новый активный *Процесс* из тех которые получили сообщение *возобновить*. Единственный экземпляр класса *Планировщик исполнителя* содержит все *Процессы* которые получили сообщение *возобновить*. Этот экземпляр *Планировщика исполнителя* именуется глобально *Исполнитель*. Активный *Процесс* можно получить послав *Исполнителю* сообщение *активный процесс*. Например, активный *Процесс* можно завершить при помощи выражения:

```
Исполнитель активный процесс завершить.
```

Это будет последним предложением выполненным данным *Процессом*. Любое предложение следующее в методе никогда не будет выполнено. Также *Исполнитель* может завершить активный *Процесс* в ответ на сообщение *завершить активный*.

```
Исполнитель завершить активный.
```

15.1.2 Приоритеты

Обычно, планирование использования *Процессом* исполнителя происходит на основе простого правила: первым пришёл первым обслужен. При получении активным *Процессом* сообщения *приостановить* или *завершить*, активным становится *Процесс* который

ждал наибольшее время. Чтобы предоставить больший контроль над тем какой *Процесс* запустить, *Исполнитель* использует очень простой механизм приоритетов. Существует фиксированное количество уровней приоритета пронумерованных возрастающими целыми значениями. *Процесс* с более высоким приоритетом получает исполнителя до *Процесса* с более низким приоритетом, не зависимо от порядка в котором он был запрошен. При создании *Процесса* (при помощи сообщения *разветвить* или *новый процесс*), он получает тот же приоритет что и создавший его *Процесс*. Приоритет процесса может быть изменён посылкой сообщения *приоритет:* с приоритетом в качестве аргумента. Либо приоритет *Процесса* может быть задан когда он разветвляется при помощи сообщения *разветвить от:* с приоритетом в качестве аргумента. Например, рассмотрим следующие предложения выполняемые *Процессом* с приоритетом 4.

```
процесс слова ← [
  ['now' показать в: 50 @ 100.] разветвить от: 6.
  ['is' показать в: 100 @ 100.] разветвить от: 5.
  'the' показать в: 150 @ 100.]
    новый процесс.
процесс слова приоритет: 7.
'time' показать в: 200 @ 100.
процесс слова возобновить.
'for' показать в: 250 @ 100.
```

Последовательность вывода на экран слов будет следующей.

```
             time
         the time
now      the time
now  is  the time
now  is  the time  for
```

Приоритеты задаются при помощи сообщений *Процессам* и *Контекстам блоков*.

Протокол экземпляров *Процесса*

доступ

приоритет: **целое**
Присваивает приоритету получателя целое.

Протокол экземпляров *Контекста блока*

планирование

разветвить от: **приоритет**
Создаёт новый процесс выполнения предложений содержащихся в получателе. Планируется выполнение нового процесса с уровнем приоритета приоритет.

В действительности методы системы Смолток не задают приоритеты с помощью литералов целых. Подходящий приоритет всегда получается при помощи посылки сообщения *Исполнителю*. Сообщения используемые для получения приоритетов показаны в протоколе класса *Планировщик исполнителя*.

Другое сообщение *Исполнителя* позволяет получить доступ к исполнителю другим *Процессам*, с тем же приоритетом что и у активного *Процесса*. *Планировщик исполнителя* отвечает на сообщение *уступить* приостановкой активного *Процесса* и помещением его в конец ожидающих *Процессов* с его приоритетом. Затем первый *Процесс* списка становится активным *Процессом*. Если нету других *Процессов* с тем же приоритетом, то сообщение *уступить* ничего не делает.

Протокол экземпляров *Планировщика исполнителя*

доступ

активный приоритет
Возвращает приоритет текущего выполняющегося процесса.

активный процесс
Возвращает текущий выполняющийся процесс.

изменение состояния процесса

завершить активный
Завершает текущий выполняющийся процесс.

уступить
Даёт возможность выполниться другому процессу, с тем же приоритетом что и у текущего выполняющегося процесса.

имена приоритетов

высокий приоритет ВВ
Возвращает приоритет с которым должны выполняться наиболее критичные ко времени процессы ввода/вывода.

низкий приоритет ВВ
Возвращает приоритет с которым должны выполняться большинство процессов ввода/вывода.

приоритет фона системы
Возвращает приоритет с которым должны выполняться фоновые процессы системы.

приоритет синхронных процессов
Возвращает приоритет с которым должны выполняться процессы отслеживающие реальное время.

приоритет фона пользователя
Возвращает приоритет с которым должны выполняться фоновые процессы созданные пользователем.

приоритет прерываний пользователя
Возвращает приоритет с которым должны выполняться процессы созданные пользователем которые требуют немедленного обслуживания.

приоритет интерфейса с пользователем

Возвращает приоритет с которым должны выполняться процессы интерфейса с пользователем.

Сообщения запрашивающие приоритеты у *Планировщика исполнителя* были перечислены выше в алфавитном порядке т.к. это стандартный способ представления описания протокола. Теже сообщения перечислены ниже начиная с высшего приоритета к низшему вместе с некоторыми примерами *Процессов* которые могут иметь этот приоритет.

приоритет синхронных процессов	Процесс следящий за часами реального времени (смотри описание класса "'Wakeup"' ниже в этой главе).
высокий приоритет ВВ	Процесс следящий за устройством ввода/вывода локальной сети.
низкий приоритет ВВ	Процесс следящий за устройствами ввода пользователя и *Процесс* распределяющий пакеты из локальной сети.
приоритет прерываний пользователя	Любой *Процесс* разветвлённый интерфейсом с пользователем который должен быть запущен немедленно.
приоритет интерфейса с пользователем	*Процесс* выполняющий действия заданные через интерфейс с пользователем (редактирование, просмотр, программирование и отладка).
приоритет фона пользователя	Любой *Процесс* разветвлённый интерфейсом с иользователем который должен быть выполнен только тогда когда ничего другого не происходит.
приоритет фона системы	*Процессы* системы которые должны выполняться когда не происходит ничего другого.

15.2 Семафоры

Последовательность действий представляемая *Процессом* выполняется асинхронно с действиями представленными другими *Процессами*. Работа одного *Процесса* независима от работу другого *Процесса*. Это справедливо для *Процессов* которые никогда не взаимодействуют. Например, два *Процесса* приведённых ниже которые показывают часы и сортируют набор, наверно, совсем не нуждаются во взаимодействии между собой.

```
[
  Восточное время показать.
  Горное время показать.
  Тихоокеанское время показать. ]
    разветвить.
алфавитный список ← список имён сортированный.
```

Однако некоторые, в значительной степени независимые, *Процессы* иногда должны взаимодействовать. Действия таких не жёстко связанных *Процессов* должны быть синхронизованы при взаимодействии. Экземпляры *Семафора* предоставляют простую форму обеспечения синхронных взаимодействий между *Процессами* независимыми в остальное время. *Семафор* предоставляет простое синхронное сообщение (~1 бита информации) от одного процесса другому. *Семафор* предоставляет возможность выполнить пассивное ожидание *Процессу* который старается получить сигнал который ещё не был послан. *Семафоры* это единственный безопасный механизм предоставляемый для взаимодействий между *Процессами*. Любые другие механизмы взаимодействия должны пользоваться *Семафорами* для обеспечения синхронизации.

Взаимодействие через *Семафор* начинается одним *Процессом* с посылки ему сообщения *сигнал*. На другом взаимодействующем конце, другой *Процесс* ждёт получения связи при помощи посылки сообщения *ждать* тому же *Семафору*. Не имеет значения в каком порядке посланы эти два сообщения, выполнение *Процесса* ждущего получения сигнала не будет продолжено до его посылки. Управление из сообщения ждать возвратится *Семафором* столько раз сколько раз он получил сообщение *сигнал*. Если семафору по-

слано одно сообщение *сигнал* и два сообщения *ждать*, то он не вернёт управление из одного сообщения *ждать*. При получении *Семафором* сообщения ждать для которого не было послано соответствующего сигнала, он приостанавливает процесс из которого было послано сообщение *ждать*.

Протокол экземпляров *Семафора*

связь

сигнал
Посылает сигнал через получателя. Если один или более *Процессов* были приостановлены из за ожидания сигнала, позволяет одному процессу, который ждал больше всего, продолжиться. Если ждущих процессов нет, то запоминается избыточный сигнал.

ждать
Перед тем как продолжить выполнение активный процесс должен получить сигнал через получателя. Если сигнала не было послано, то активный процесс будет приостановлен до посылки сигнала.

Процессы которые были приостановлены будут возобновлены в том же порядке в котором они были приостановлены. Приоритет *Процесса* рассматривается *Планировщиком исполнителя* только при планировании использования им исполнителя. Каждый *Процесс* ждёт пока *Семафор* возобновит его на основе порядка первым пришёл — первым обслужен, не зависимо от приоритета. *Семафор* позволяет *Процессу* ждать ещё не посланного сигнала без использования времени исполнителя. *Семафор* не возвращает управление из сообщения *ждать* до тех пор пока не будет послан сигнал. Одно из главных преимуществ создания независимых процессов для выполнения действий заключается в том что если процессу нужно что-то не доступное в данный момент, то другой процесс может продолжать выполняться в то время как первый процесс ждёт появления ресурса. Примерами того в чём процесс может нуждаться и что может быть недоступно являются: устройства, события от пользователя (нажатие кнопки или движение указывающего устройства) и разделяемые структуры данных. Определённое время дня также

может быть тем в чём нуждается процесс для продолжения выполнения.

15.2.1 Взаимное исключение

Семафоры могут использоваться для обеспечения взаимного исключения использования некоторого ресурса различными *Процессами*. Например, *Семафор* можно использовать для создания структуры данных безопасной для доступа различным *Процессами*. Следующее определение простой структуры данных типа первым пришёл первым ушёл не обеспечивает взаимное исключение обращений к ней.

имя класса Простая очередь
надкласс Объект
имена переменных экземпляра ряд содержимое пположение чтения пположение записи

методы класса

создание экземпляра

```
новый
  ↑сам новый: 10.
новый: размер
  ↑над новый ини: размер.
```

методы экземпляра

доступ

```
следующий
  | значение |
  положение чтения = положение записи
    истина: [сам ошибка: 'empty queue'.]
    ложь: [
      значение ← ряд содержимое от: положение чтения.
      ряд содержимое от: положение чтения пом: пусто.
      положение чтения ← положение чтения + 1.
```

```
        ↑значение. ].
пом следующим: значение
  положение записи > ряд содержимое размер
    истина: [сам создать пространство для записи. ].
  ряд содержимое от: положение записи пом: значение.
  положение записи ← положение записи + 1.
  ↑значение.
размер
  ↑положение записи − положение чтения.
```

проверки

```
пустой
  ↑положение записи = положение чтения.
```

собственные

```
ини: размер
  ряд содержимое ← Ряд новый: размер.
  положение чтения ← 1.
  положение записи ← 1.
создать пространство для записи
  | размер содержимого |
  положение чтения = 1
    истина: [ряд содержимое расти. ]
    ложь: [
     размер содержимого ← положение записи − положение чтения.
      1
        до: размер содержимого
        делать: [
          :номер |
          ряд содержимое
            от: номер
        пом: ( ряд содержимое от: номер + положение чтения − 1 ). ].
      положение чтения ← 1.
      положение записи ← размер содержимого + 1. ].
```

Простая очередь запоминает своё содержимое в *Ряде* с именем *ряд содержимое* и содержит два номера с именами *положение чтения* и *положение записи*. Новое содержимое добавляется в *положение записи* и удаляется из *положения чтения*. Собственное сообщение *создать пространство для записи* посылается когда не остаётся места в конце *ряда содержимого* для запоминания нового объекта. Если *ряд содержимое* полностью заполнен, то его размер увеличивается. Иначе, содержимое перемещается в начало *ряда содержимого*.

Проблема с посылкой сообщений *Простой очереди* из различных *Процессов* заключается в том что больше одного *Процесса* одновременно могут выполнять метод *следующий* или *пом следующим:*. Допустим *Простой очереди* было послано сообщение *следующий* из одного *Процесса*, и в данный момент выполняется предложение:

```
значение ← ряд содержимое от: положение чтения.
```

В это время просыпается *Процесс* с более высоким приоритетом и посылает другое сообщение *следующий* той же самой *Простой очереди*. Т.к. *положение чтения* не было увеличено, то второе выполнение вышеприведённого предложения присвоит тот же объект *значению*. *Процесс* с более высоким приоритетом удалит ссылку на объект из *ряда содержимого*, увеличит *положение чтения* и возвратит удалённый объект. Когда *Процесс* с более низким приоритетом получит управление, *положение чтения* будет увеличено поэтому он удалит ссылку на следующий объект из *ряда содержимого*. Этот объект должен был быть значением сообщения *следующий*, но был отброшен и оба сообщения *следующий* вернули один и тот же объект.

Чтобы гарантировать взаимное исключение каждый *Процесс* должен ждать один и тот же *Семафор* перед использованием ресурса и затем сигналить *Семафором* после его использования. Следующий подкласс *Простой очереди* предоставляет взаимное исключение поэтому его экземпляры могут быть использованы различными *Процессами*.

имя класса Простая разделяемая очередь

```
надркласс Простая очередь
имена переменных экземпляра защита доступа
```

методы экземпляра

доступ

```
следующий2
  | значение |
  защита доступа ждать.
  значение ← над следующий.
  защита доступа сигнал.
  ↑ значение.
пом следующим2: значение
  защита доступа ждать.
  над пом следующим: значение.
  защита доступа сигнал.
  ↑ значение.
```

собственные

```
ини2: размер
  над ини: размер.
  защита доступа ← Семафор новый.
  защита доступа сигнал.
```

Т.к. взаимное исключение это обычное использование *Семафоров*, то он содержит сообщение для этого. Селектор этого сообщения *критический:*. Ниже приведена реализация этого метода.

```
критический: блок взаимного исключения
  | значение блока |
  сам ждать.
  [значение блока ← блок взаимного исключения значение.]
    гарантировать: [сам сигнал.].
  ↑ значение блока.
```

Семафор использующийся для взаимного исключения должен начинать работу с одним полученным сигналом чтобы первый *Процесс* мог войти в критическую область. Класс *Семафор* предостав-

ляет специальное инициализирующее сообщение, для взаимного исключения, которое посылает один сигнал новому экземпляру.

Протокол экземпляров *Семафора*

взаимное исключение

критический: блок
Выполняет блок когда не выполняется другого критического блока.

Протокол класса *Семафор*

создание экземпляра

для взаимного исключения
Возвращает новый *Семафор* с одним излишним сигналом.

Реализация *Простой разделяемой очереди* должна быть изменена так:

имя класса Простая разделяемая очередь
надркласс Простая очередь
имена переменных экземпляра защита доступа

методы экземпляра

доступ

следующий
```
| значение |
защита доступа критический: [значение ← над следующий.].
↑значение.
```
пом следующим: значение
```
защита доступа критический: [над пом следующим: значение.].
↑значение.
```

собственные

ини: размер
над ини: размер.
защита доступа ← Семафор для взаимного исключения.

15.2.2 Разделяемые ресурсы

Чтобы разделять ресурс между двумя *Процессами* не достаточно только обеспечить взаимно исключающий доступ к нему. *Процессы* должны иметь возможность узнавать о доступности ресурса. *Простая разделяемая очередь* нормально работает с множественным доступом к ней, но если будет произведена попытка удалить объект из пустой *Простой разделяемой очереди*, то возникнет ошибка. В среде с асинхронными *Процессами*, трудно гарантировать что попытки удаления объектов (при помощи сообщения *следующий*) будут производиться только после того как они были добавлены (при помощи сообщения *пом следующим:*). Однако, *Семафоры* также используются чтобы сигнализировать доступность разделяемых ресурсов. Семафор представляющий ресурс сигналит после того как становится доступным каждая единица ресурса и ждёт перед потреблением каждой единицы. Поэтому, если производится попытка получить ресурс до того как он был произведён, потребитель просто ждёт.

Класс *Безопасная разделяемая очередь* это пример использования *Семафоров* в качестве средства сообщения о доступности ресурса. *Безопасная разделяемая очередь* подобна *Простой разделяемой очереди*, но она использует для представления доступности содержимого очереди ещё один *Семафор* с именем *значение доступно*. *Безопасная разделяемая очередь* не является частью системы Смолток, она здесь описана только как пример. *Разделяемая очередь* это класс который в действительности используется для связи между процессами системы. *Разделяемая очередь* предоставляет функциональность подобную предоставляемой *Безопасной разделяемой очередью*. Протокол *Разделяемой очереди* будет дан ниже в этой главе.

имя класса Безопасная разделяемая очередь

```
надркласс Простая очередь
имена переменных экземпляра значение доступно защита доступа
```

методы экземпляра

доступ

```
следующий
  | значение |
  значение доступно ждать.
  защита доступа критический: [значение ← над следующий.].
  ↑значение.
пом следующим: значение
  защита доступа критический: [над пом следующим: значение.].
  значение доступно сигнал.
  ↑значение.
```

собственные

```
ини: размер
  над ини: размер.
  защита доступа ← Семафор для взаимного исключения.
  значение доступно ← Семафор новый.
```

15.2.3 Прерывания аппаратуры

Также экземпляры *Семафора* используются для связи между аппаратурой и *Процессами*. При этом использовании, семафор замещает прерывание в значении информирования об изменении состояния оборудования. Виртуальная машина Смолтока задаёт три условия при которых семафором посылаются сигналы.

- событие от пользователя: была нажата кнопка на клавиатуре, на указывающем устройстве или указывающее устройство перемещено.
- таймаут: было достигнуто определённое значение миллисекундными часами.

- мало места: доступная память объектов стала меньше определённого предела.

Эти три *Семафора* соответствуют трём процессам следящим за событиями от пользователя, миллисекундными часами и использованием памяти. Каждый следящий *Процесс* посылает сообщение *ждать* соответствующему *Семафору* что приостанавливает его до тех пор пока не произойдёт что то интересное. Когда *Семофор* просигналит, то *Процесс* будет возобновлён. Виртуальная машина уведомляется об ожидании этих трёх типов событий при помощи примитивных методов.

Класс *Пробуждение* это пример использования *Семафора* который сигналит об одном из таких событий. *Пробуждение* предоставляет сервис будильника для *Процесса* при помощи слежения за часами. Класс *Пробуждение* не является частью система Смолток; он описан здесь только в качестве примера. В системе Смолток слежением за миллисекундными часами занимается класс *Задержка*. Этот класс предоставляет функциональность подобную функциональности *Пробуждения*. Описание протокола *Задержки* будет дано ниже в этой главе. *Пробуждение* предоставляет сообщение которое приостанавливает пославший его *Процесс* на заданное количество миллисекунд. Следующее предложение приостанавливает *Процесс* на три четверти секунды.

```
Пробуждение спустя: 750.
```

Когда *Пробуждение* получает сообщение *спустя:* оно создаёт новый экземпляр который запоминает значение часов при котором должно произойти пробуждение. Новый экземпляр содержит *Семафор* на котором активный *Процесс* должен ждать до достижения времени пробуждения. *Пробуждение* хранит все свои экземпляры в списке отсортированном по времени пробуждения. *Процесс* следит за миллисекундными часами виртуальной машины по наиболее раннему из времён пробуждения и позволяет соответствующему приостановленному *Процессу* продолжиться. Этот *Процесс* создаётся методом класса *инициализировать процесс синхронизации*. На *Семафор* используемый для слежения за часами ссылается переменная класса *Семафор синхронизации*. Виртуальная машина уведомляется о том что нужно следить за часами при помощи следу-

ющего предложения находящегося в методе экземпляра *следующее пробуждение*.

```
Исполнитель
    прим сигнал: Семафор синхронизации
    при миллисекундах: время пробуждения.
```

На список экземпляров ждущих продолжения ссылается переменная класса *Ожидающие пробуждения*. Есть ещё один *Семафор* с именем *Защита доступа* который предоставляет взаимное исключение доступа к *Ожидающим пробуждениям*.

имя класса Пробуждение
надркласс Объект
имена переменных экземпляра время будильника симафор будильник
имена переменных класса Ожидающие пробуждения Защита доступа Семафор синхронизации

методы класса

сервис будильника

```
спустя: количество миллисекунд
    (сам новый продолжительность сна: количество миллисекунд)
ждать пробуждения.
```

инициализация класса

```
инициализировать
    Семафор синхронизации ← Семафор новый.
    Защита доступа ← Семафор для взаимного исключения.
    Ожидающие пробуждения ← Сортированный набор новый.
    сам инициализировать синхронный процесс.
инициализировать синхронный процесс
    [
        [истина.]
            пока истина: [
                Семафор синхронизации ждать.
                Защита доступа ждать.
```

```
            Ожидающие пробуждения удалить первый проснуться.
            Ожидающие пробуждения пустой
             ложь: [Ожидающие пробуждения первый следующее пробуждение.].
            Защита доступа сигнал.].]
        разветвить от: Исполнитель приоритет синхронных процессов.
```

методы экземпляра

задержка процесса

ждать пробуждения

```
  Защита доступа ждать.
  Ожидающие пробуждения добавить: сам.
   Ожидающие пробуждения первый == сам истина: [сам следующее пробуждение.].
  Защита доступа сигнал.
  семафор будильник ждать.
```

сравнение

< другое пробуждение

```
  ↑время будильника < другое пробуждение время пробуждения.
```

доступ

время пробуждения

```
  ↑время будильника.
```

собственные

следующее пробуждение

```
  Исполнитель
    прим сигнал: Семафор синхронизации
    при миллисекундах: время будильника.
```

продолжительность сна: количество миллисекунд

```
  время будильника ← Время значение часов в миллисекундах + количество миллисекунд.
  семафор будильник ← Семафор новый.
```

проснуться
семафор будильник сигнал.

15.3 Класс *Разделяемая очередь*

Класс *Разделяемая очередь* это класс системы чьи экземпляры являются безопасным средством сообщения между *Процессами*. Его протокол и реализация похожи на протокол примера *Безопасная разделяемая очередь* который был приведён раньше в этой главе.

Протокол экземпляров *Разделяемой очереди*

доступ

следующий
Возвращает первый объект который был добавлен к получателю и ещё не был удалён. Если получатель пуст, то текущий *Процесс* приостанавливается до тех пор пока в получателя не будет добавлен объект.

пом следующим: значение
Добавляет значение к содержимому получателя. Если *Процесс* был приостановлен для ожидания объекта, то ему позволяется возобновиться.

15.4 Класс *Задержка*

Задержка позволяет *Процессу* приостановиться на заданное время. *Задержка* создаётся заданием продолжительности в течении которой активный *Процесс* будет приостановлен.

```
полуминутная задержка ← Задержка на секунды: 30.
короткая задержка ← Задержка на миллисекунды: 50.
```

Простое создание *Задержки* не имеет эффекта на активный *Процесс*. *Задержка* приостанавливает активный *Процесс* в ответ на сообщение *ждать*. Каждое из следующих предложений приостановит выполнение активного *Процесса* на 30 секунд.

```
полуминутная задержка ждать.
```

(Задержка на секунды: 30) ждать.

Протокол класса *Задержка*

создание экземпляра

на миллисекунды: **количество миллисекунд**
Возвращает новый экземпляр который будет приостанавливать активный *Процесс* на количество миллисекунд когда ему будет посылаться сообщение ждать.

на секунды: **количество секунд**
Возвращает новый экземпляр который будет приостанавливать активный *Процесс* на количество секунд когда ему будет посылаться сообщение *ждать*.

Протокол экземпляров *Задержки*

доступ

время возобновления
Возвращает значение миллисекундных часов при котором приостановленный *Процесс* будет возобновлён.

задержка процесса

ждать
Приостанавливает выполнение активного *Процесса* до тех пока миллисекундные часы не не достигнут подходящего значения.

При помощи следующего предложения могут быть созданы виртуальные часы.

```
[
  [истина.]
    пока истина: [
      Время текущее цепь для печати показать в: 100@100.
      (Задержка на секунды: 1) ждать.].]
    разветвить.
```

На мониторе будет показываться текущее время каждую секунду.

Глава 16

Протокол Классов

Оглавление

Сейчас введён протокол для большинства классов системы которые описывают основные компоненты системы Смолток. Одним значительным исключением является протокол для самих классов. Четыре класса: *Поведение*, *Описание класса*, *Метакласс* и *Класс* — вместе предоставляют средства для описания новых классов. Создание нового класса включает компиляцию методов и задание имён для переменных экземпляра, переменных класса, переменных пула и для самого класса.

В главах 3, 4 и 5 были введены основные концепции представляемые этими классами. Подводя итог можно сказать что Смолток программист задаёт новый класс путём создания подкласса другого класса. Например, класс *Набор* это подкласс *Объекта*; класс *Ряд* это подкласс *Набора ряда* (чья цепь наследования заканчивается *Объектом*).

1. Каждый класс в конечном счёте является подклассом *Объекта*, за исключением самого *Объекта*, у которого нету надкласса. В частности, *Класс* это подкласс *Описания класса*, который является

Объект

Величина
Знак
Дата
Время

Число
Плавающее
Дробь
Целое
Большое положительное целое
Большое отрицательное целое
Малое целое

Ключ поиска
Ассоциация

Связь
Процесс

Набор
Набор последовательность
Связанный список

Семафор

Набор ряд
Ряд

Растровое изображение

Ряд серий
Цепь
Символ
Текст
Ряд байтов

Интервал
Упорядоченный набор
Сортированный набор
Мешок
Набор отображение
Множество
Словарь
Тождественный словарь

Поток
Позиционируемый поток
Поток чтения
Поток записи
Поток чтения записи
Поток файл

Случайное число

Неопределённый объект
Логика
Истина
Ложь

Планировщик исполнителя
Задержка
Разделяемая очередь

Поведение
Описание класса
Класс
Метакласс

Точка
Прямоугольник

подклассом *Поведения* который является подклассом *Объекта*.

В системе есть два вида объектов, один который может создавать свои экземпляры (классы) и другой который не может.

2. Каждый объект это экземпляр класса.

Каждый класс сам является экземпляром класса. Класс класса называется его метаклассом.

3. Каждый класс это экземпляр метакласса.

На метакласс нельзя сослаться по имени как на другие классы. Вместо этого на них ссылаются при помощи унарного сообщения *класс* посылаемого экземпляру метакласса. Например, на метакласс *Набора* ссылаются так: *Набор класс*; ссылка на метакласс *Класса* это *Класс класс*.

В системе Смолток при создании нового класса автоматически создаётся метакласс. У метакласса имеется только один экземпляр. Методы из категории «методы класса», в описании класса, находятся в метаклассе класса. Это следует из способа которым ищутся методы; когда сообщение посылается объекту, то поиск соответствующего метода начинается в классе объекта. Например, когда сообщение посылается *Словарю*, то поиск начинается в метаклассе *Словаря*. Если метод не находится в метаклассе, то поиск продолжается в надклассе метакласса. В данном случае надкласс это *Множество класс*, метакласс надкласса *Словаря*. Если требуется, то поиск продолжается по цепи наследования вплоть до метакласса *Объект класс*.

В диаграммах этой главы все стрелки со сплошными линиями означают отношение подкласса; стрелки со штриховыми линиями это отношение экземпляра. A —> B означает что A это экземпляр B. Сплошные серые линии указывают ирархию классов; сплошные чёрные линии указывают ирархию метаклассов.

Т.к. цепь надклассов всех объектов заканчивается *Объектом*, как показано на рисунке 16.1, и у *Объекта* нету надкласса, то надкласс метакласса *Объекта* не определяется правилом поддержания параллельной ирархии. В этом месте появляется *Класс*. надкласс метакласса *Объект класс* это *Класс*.

4. Все метаклассы (в конце концов) это подклассы *Класса* (рисунок 16.2).

Т.к. метаклассы это объекты, то они должны быть экземплярами

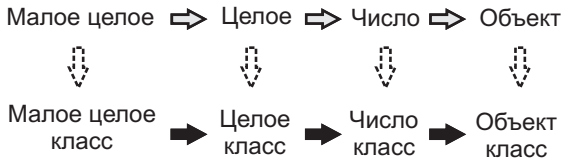


Рис. 16.1

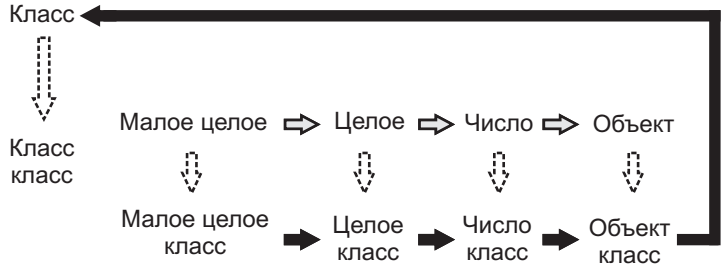


Рис. 16.2

класса. Каждый метакласс это экземпляр *Метакласса*. Сам *Метакласс* это экземпляр *Метакласса*. Это место зацикливания системы — метакласс *Метакласса* должен быть экземпляром *Метакласса*.

5. Каждый метакласс это экземпляр *Метакласса* (рисунок 16.3).

Рисунок 16.4 показывает взаимоотношения между *Классом*, *Описанием класса*, *Поведением* и *Объектом*, и соответствующих им метаклассов. Ирархия классов идёт до *Объекта*, а ирархия метаклассов следует через *Объект класс* до *Класса* и до *Объекта*. В то время как методы *Объекта* поддерживают поведение общее для всех объектов, методы *Класса* и *Метакласса* поддерживают поведение общее для всех классов.

6. Методы *Класса* и его надклассов поддерживают поведение общее для тех объектов которые являются классами.

7. Методы экземпляров *Метакласса* добавляют поведение специфичное для отдельного класса.

Соответствие между ирархиями классов и метаклассов показано на рисунке 16.5.

16.1 Класс *Поведение*

Класс *Поведение* определяет минимальное состояние необходимое объектам у которых есть экземпляры. В частности, *Поведение* определяет состояние используемое интерпретатором Смолтока. Он предоставляет основной интерфейс компилятору. Состояние описываемое *Поведением* включает связь ирархии классов, словарь методов и описание экземпляров в терминах количества и представления их переменных.

Протокол сообщений класса *Поведение* будет описан в четырёх категориях — создание, доступ, проверки и перебор. Эти категории и их подкатегории, как описано ниже, предоставляют модель для размышления о функциональности классов системы Смолток.

Наброcок протокола всех классов
создание

- создание словаря методов
- создание экземпляров
- создание ирархии классов

доступ

- доступ к содержимому словаря методов
- доступ к экземплярам и переменным: экземпляра, класса и пула
- доступ к ирархии класса

проверки

- проверка содержимого словаря методов
- проверка формы экземпляров
- проверка ирархии класса

перебор

- перебор подклассов и экземпляров

Протокол создание класса *Поведение*

Методы в описании класса хранятся в словаре с именем словарь методов. Иногда, он также называется словарь сообщений. Ключями этого словаря являются селекторы; значения это откомпилированная форма методов (экземпляры *Откомпилированного метода*). Протокол создания словаря методов поддерживает компиляцию методов и также добавление ассоциации между селектором и откомпилированным методом. Также поддерживается доступ и к откомпилированным и не откомпилированным (исходным) версиям методов.

Протокол экземпляров *Поведения*

создание словаря методов

словарь методов: **словарь**
Запоминает аргумент, словарь, в качестве словаря методов получателя.

добавить селектор: **селектор** **с методом:** **откомпилированный метод**
Добавляет селектор сообщения, селектор, с соответствующим откомпилированным методом, откомпилированный метод, в словарь методов получателя.

удалить селектор: **селектор**
Удаляет аргумент, селектор (который является *Символом* представляющим селектор сообщения), из словаря методов получателя. Если селектора нету в словаре методов, то сообщается об ошибке.

компилировать: **текст**
Аргумент, текст, это либо *Цепь* либо объект который можно преобразовать в *Цепь* либо это *Позиционируемый поток* для доступа к объекту являющемуся *Цепью*. Компилирует текст как исходный текст в контексте переменных получателя. Сообщает об ошибке если текст не может быть скомпилирован.

компилировать: текст уведомлять: запрощик
Компилирует аргумент, текст, и вставляет результат в словарь методов получателя. Если происходит ошибка, то посылает соответствующее сообщение аргументу, запрощик.

перекомпилировать: селектор
Компилирует метод связанный с селектором сообщения, селектор.

декомпилировать: селектор
Находит откомпилированный код связанный с аргументом, селектор, и декомпилирует его. Возвращает полученный исходный текст в виде *Цепи*. Если селектора нету в словаре методов, то сообщается об ошибке.

компилировать весь
Компилирует все методы из словаря методов получателя.

Экземпляры классов создаются при помощи посылки сообщений *новый* или *новый:*. Эти два сообщения могут быть переопределены в словаре методов метакласса для добавления особого поведения инициализации. Назначение любого особого инициализирования в том чтобы гарантировать что экземпляр создастся с переменными которым присвоены соответствующие экземпляры. Эта идея была продемонстрирована во всех предыдущих главах.

Допустим класс переопределяет метод новый и затем один из его подклассов хочет сделать то же так чтобы избежать изменения поведения своего надкласса. Метод из первого класса может быть таким:

```
новый
  ↑над новый присвоить переменные.
```

где сообщение присвоить переменные предоставляется протоколом экземпляра этого класса. При посылке сообщения новый псевдо переменной *супер* вызывается метод создания экземпляра заданный в классе *Поведение*; затем результату, новому экземпляру, посылается сообщение *присвоить переменные*. В подклассе нельзя использовать сообщение *супер новый* чтобы вызвать метод *Поведения*, т.к. оно вызовет метод первого класса. Чтобы вызвать основной метод создания экземпляра из *Поведения*, подкласс может использовать выражение *сам основной новый*. Сообщение *основной новый*

это примитивное сообщение создания экземпляра которое не должно переопределяться ни каким подклассом. В *Поведении* и *новый* и *основной новый* идентичны. *Поведением* также предоставляется подобная пара сообщений *новый:* и *основной новый:* для создания объектов переменной длины. (Заметьте что техника двойных сообщения также используется в классе *Объект* для методов доступа *от:* и *от:пом:*.)

Протокол экземпляров *Поведения*

создание экземпляра

новый
Возвращает экземпляр получателя без нумерованных переменных. Посылает получателю сообщение *новый: 0* если получатель нумерованный.

основной новый
Такой же как и новый, за исключением того что этот метод не должен переопределяться подклассами.

новый: целое
Возвращает экземпляр получателя с количеством нумерованных переменных целое. Сообщает об ошибке если получатель не нумерованный.

основной новый: целое
Такой же как и *новый:*, за исключением того что этот метод не должен переопределяться подклассами.

Протокол создания классов включает сообщения для помещения класса в ирархию классов системы. Т.к. ирархия линейна, то нужны только сообщения для задания надкласса и добавления или удаления подклассов.

Протокол экземпляров *Поведения*

создание ирархии классов

надкласс: класс
Присваивает надклассу получателя аргумент, класс.

addSubclass: aClass

Make the argument, aClass, be a subclass of the receiver.

removeSubclass: **aClass**

Remove the argument, aClass, from the subclasses of the receiver.

Несмотря на то что протокол создания *Поведения* позволяет писать предложения создающие новые описания классов, обычно используются преимущества графического окружения в которое встроен язык Смолток, и предоставляется интерфейс в котором пользователь заполняет графические формы чтобы задать информацию о различных частях класса.

Протокол доступ класса *Поведение*

Сообщения которые обращаются к содержимому словаря методов делятся на те которые служат для обращения к локальному словарю методов класса, и те которые служат для доступа к словарям методов класса и всех его надклассов.

Протокол экземпляров *Поведения*

доступ к словарю методов

селекторы

Возвращает *Множество* всех селекторов сообщений определённых в локальном словаре методов получателя.

все селекторы

Возвращает *Множество* всех селекторов сообщений которые может понимать экземпляр получателя. Оно содержит все сообщения селекторов из словаря методов получателя и каждого из надклассов получателя.

откомпилированный метод от: **селектор**

Возвращает откомпилированный метод связанный с аргументом, селектор, селектором сообщения локального словаря методов получателя. Если селектор не найден, то сообщается об ошибке.

исходный текст от: **селектор**

Возвращает *Цепь* которая является исходным текстом связанным с аргументом, селектор, селектором сообщения локального словаря методов получателя. Если селектор не найден, то сообщается об ошибке.

исходный метод от: **селектор**

Возвращает *Текст* для исходного текста связанного с аргументом, селектор, селектором сообщения локального словаря методов получателя. Этот *Текст* содержит выделение жырным шрифтом образца сообщения. Если селектор не найден, то сообщается об ошибке.

У экземпляра может быть переменные экземпляра, нумерованные переменные экземпляра, переменные класса и словари переменных пула. Опять же, различие между локально заданными переменными и переменными унаследованными от надклассов отражается в протоколе доступа.

Протокол экземпляров *Поведения*

доступ к экземплярам и переменным

все экземпляры

Возвращает *Множество* всех прямых экземпляров получателя.

некоторый экземпляр

Возвращает существующий экземпляр получателя.

количество экземпляров

Возвращает количество существующих в данный момент экземпляров получателя.

имена переменных экземпляра

Возвращает *Ряд* имён переменных экземпляра заданных в получателе.

имена переменных экземпляра подклассов

Возвращает *Множество* имён переменных экземпляра заданных в подклассах получателя.

все имена переменных экземпляра

Возвращает *Ряд* из имён переменных экземпляра получателя которые заданы в получателе и всех его надклассах. Упорядочение *Ряда* это порядок в котором переменные используются интерпретатором Смолтока.

имена переменных класса
Возвращает *Множество* имён переменных класса заданных в получателе локально.

все имена переменных класса
Возвращает *Множество* имён переменных класса заданных в получателе и его надклассах.

разделяемые пулы
Возвращает *Множество* имён пулов (словарей) которые заданы в получателе локально.

все разделяемые пулы
Возвращает *Множество* имён пулов (словарей) которые заданы в получателе и каждом из его надклассов.

Поэтому:

предложение	**результат**
Упорядоченный набор имена переменных экземпляра.	#('ряд' 'первый номер' 'последний номер')
Упорядоченный набор имена переменных экземпляра подклассов.	#('сортирующий блок')
Сортированный набор все имена переменных экземпляра.	#('ряд' 'первый номер' 'последний номер' 'сортирующий блок')
Текст разделяемые пулы.	an OrderedCollection(a Dictionary(size 110))

Протокол доступа включает сообщения для получения наборов надклассов и подклассов класса. Эти сообщения разделяются на две группы: непосредственные подклассы и надклассы класса и все классы из цепи наследования класса.

Протокол экземпляров *Поведения*

доступ к ирархии классов

подклассы
Возвращает *Множество* содержащие непосредственные подклассы получателя.

все подклассы
Возвращает *Множество* содержащие подклассы получателя и их потомков.

со всеми подклассами
Возвращает *Множество* из получателя, подклассов получателя и потомков подклассов получателя.

надкласс
Возвращает непосредственный надкласс получателя.

все надклассы
Возвращает *Упорядоченный набор* надклассов получателя. Первый элемент это непосредственный надкласс получателя, затем идёт его надкласс и т.д.; последний элемент это всег *Прото объект*.

Поэтому:

предложение	**результат**
Цепь надкласс.	ArrayedCollection
Набор ряд подклассы.	{DirectoryEntry . SoundBuffer . WordArray . RunArray . Array . ShortIntegerArray . Bitmap . FloatArray . Text . ShortRunArray . ByteArray . TranslatedMethod . IntegerArray . ColorArray . String . SparseLargeTable}

Набор ряд все подклассы.	a Set(ShortRunArray PointArray WideSymbol ByteString Array Symbol ColorArray ByteArray FixedIdentitySet DependentsArray TTFIndexToLocation Bitmap KlattFrame QuickStack TranslatedMethod ShortIntegerArray Cubic WeakActionSequence CompiledMethod UUID WAExternalID WideString RunArray a subclass of String WeakActionSequenceTrappingErrors Text String SoundBuffer SparseLargeTable WordArrayForSegment ShortPointArray FloatArray ByteSymbol IntegerArray QuickIntegerDictionary DirectoryEntry WeakArray WordArray KedamaFloatArray ActionSequence)

Набор ряд со всеми подклассами.	a Set(ShortRunArray PointArray WideSymbol ByteString Array Symbol ColorArray ByteArray FixedIdentitySet DependentsArray TTFIndexToLocation Bitmap KlattFrame QuickStack TranslatedMethod ShortIntegerArray Cubic WeakActionSequence CompiledMethod UUID WAExternalID WideString RunArray a subclass of String WeakActionSequenceTrappingErrors Text String SoundBuffer SparseLargeTable WordArrayForSegment ShortPointArray FloatArray ByteSymbol IntegerArray QuickIntegerDictionary DirectoryEntry ArrayedCollection WeakArray WordArray KedamaFloatArray ActionSequence)
Набор ряд все надклассы.	an OrderedCollection(SequenceableCollection Collection Object ProtoObject)
Набор ряд класс все надклассы.	an OrderedCollection(SequenceableCollection class Collection class Object class ProtoObject class Class ClassDescription Behavior Object ProtoObject)

Протокол проверок класса *Поведение*

Протокол проверок предоставляет сообщения служащие для нахождения информации о структуре класса и форме его экземпляров. Структура класса состоит из взаимоотношений с другими классами, его возможности отвечать на сообщения, класс в котором определено сообщения и т.д.

Содержимое словаря методов может быть проверено чтобы определить какой класс, если такой есть, реализует некоторый селектор сообщения, может ли класс отвечать на сообщение, и какие методы ссылаются на данную переменную или литерал. Все эти сообщения полезны при создании среды программирования в которой программист может изучать структуру и функциональность объектов системы.

Протокол экземпляров *Поведения*

проверки словаря методов

имеет методы
Отвечает содержит ли получатель какие-либо методы в своём (локальном) словаре методов.

содержит селектор: символ
Отвечает содержится ли сообщение с селектором равным аргументу, символ, в локальном словаре методов класса получателя.

может понимать: селектор
Отвечает может ли получаетль отвечать на сообщение чей селектор это аргумент. Селектор может находиться в словаре методов класса получателя или любом его надклассе.

который класс содержит селектор: символ
Возвращает первый класс из цепи надклассов получателя для которого аргумент, символ, является селектором метода. Возвращает пусто если ни один класс не содержит этого селектора.

которые селекторы обращаются к: имя переменной экземпляра
Возвращает *Множество* селекторов локального словаря методов чьи методы обращаются к аргументу, имя переменной экземпляра, как к именованной переменной экземпляра.

которые селекторы ссылаются на: литерал
Возвращает *Множество* селекторов чьи методы обращаются к аргументу, литерал.

в области видимости есть: имя переменной истина: блок

Определяет входит ли переменная, имя переменной, в область видимости получателя, т.е. эта переменная определена как переменная в получателе или одном из его надклассов. Если это так, то выполняется аргумент, блок.

Поэтому, например:

предложение	**результат**
Упорядоченный набор содержит селектор: #добавить первым:	истина
Сортированный набор содержит селектор: #размер	ложь
Сортированный набор может понимать: #размер	истина
Сортированный набор который класс содержит селектор: #размер	Упорядоченный набор
Упорядоченный набор которые селекторы обращаются к: #первый номер	Множество (makeRoomAtFirst before: size makeRoomAtLast insert:before: remove:ifAbsent: addFirst: first removeFirst find: removeAIISuchThat: at: atput: reverseDo: do: setIndices:)

Последнее предложение примера полезно для нахождения методов которые должны быть изменены если удалена или изменена переменная экземпляра. В дополнение к сообщениям предназначенным для доступа снаружи, *Множество* реализует все сообщения для поддержки реализации внешних сообщений.

Протокол проверок включает сообщения к классу которые отвечают как хранятся его переменные, является ли количество переменных фиксированным или переменным, и количество именованных переменных экземпляра.

Протокол экземпляров *Поведения*

проверка вида экземпляров

это указатели
Отвечает хранятся ли переменные экземпляра получателя как указатели (слова).

это биты
Отвечает хранятся ли переменные экземпляра получателя как биты (т.е. не как указатели).

это байты
Отвечает хранятся ли переменные экземпляра получателя как быйты (8-битные целые).

это слова
Answer whether the variables of instances of the receiver are stored as words.

isFixed
Answer true if instances of the receiver do not have indexed instance variables; answer false otherwise.

isVariable
Answer true if instances of the receiver do have indexed instance variables; answer false otherwise.

instSize
Answer the number of named instance variables of the receiver.

Поэтому получаем:

предложение	**результат**
LinkedList isFixed	истина
String isBytes	истина
Integer isBits	ложь
Float isWords	истина
OrderedCollection isFixed	ложь
OrderedCollection instSize	2
oc <- OrderedCollection with: $a with: $b with: $c	OrderedCollection ($a $b $c)
oc size	3

Последние четыре строки примера показываю что экземпляры *Упорядоченного набора* являются объектами переменной длины; экземпляр ун содержит три элемента. В дополнение к ним экземпляры

Упорядоченного набора упорядоченного набора имеют две именованные переменные.

В системе есть четыре вида классов. Классы которые имеют нумерованные переменные экземпляра называются классами переменной длины, а классы которые их не имеют называются классами постоянной длины. Переменные всех классов с постоянной длинной хранятся как указатели (ссылки размером со слово). Переменные классов переменной длинны могут содержать указатели, байты или слова. Т.к. указатели это ссылки размером со слово, то объект содержащий указатели должен отвечать *истина* на вопрос содержит ли он слова, но обратное не всегда верно. Сообщения инициализации определённые в *Классе* и перечисленные в следующих разделах поддерживают создания каждого вида классов.

Протокол экземпляров *Поведения*

Проверки ирархии классов

наследует от: класс
Отвечает содержится ли аргумент в цепи надклассов получателя.

вид подкласса
Возвращает *Цепь* являющуюся ключевым словом которое описывает получателя как класс: или обычный подкласс (постоянной длинны), переменный подкласс, переменный подкласс из байтов, или переменный подкласс из слов.

Поэтому:

предложение	результат
Цепь наследует от: Набор	истина
Цепь вид подкласса	переменный подкласс байтов:
Ряд вид подкласса	переменный подкласс:
Плавающее вид подкласса	переменный подкласс слов:
Целое вид подкласса	подкласс:

Протокол перебора Поведения

Сообщения определённые в классе *Поведение* также поддерживают создание множеств объектов связанных с классом и применение каждого из ник в качестве аргумента блока. Этот перебор объектов подобен тому который предоставляют классы наборов и содержит перебор всех подклассов, надклассов, экземпляров и экземпляров подклассов. В дополнение два сообщения позволяют выбрать такие подклассы или надклассы для которых блок возвращает истину.

Протокол экземпляров *Поведения*

перебор

делать для всех подклассов: **блок**
Выполняет аргумент, блок, для каждого подкласса получателя.

делать для всех надклассов: **блок**
Выполняет аргумент, блок, для каждого надкласса получателя.

делать для всех экземпляров: **блок**
Выполняет аргумет, блок, для каждого существующего экземпляра получателя.

делать для всех подэкземпляров: **блок**
Выполняет аргумент, блок, для каждого существующего экземпляра получателя и его подклассов.

выбрать подклассы: **блок**
Выполняет аргумент, блок, для каждого подкалсса получателя. Собирает в Множество только те подклассы для которых блок вернул истину. Возвращает полученное множество.

выбрать надклассы: **блок**
Выполняет аргумент, блок, для каждого надкласса получателя. Собирает в Множество только те надклассы для которых блок вернул истину. Возвращает полученное множество.

Например, чтобы понять поведение экземпляров классов наборов может быть полезно знать какой подкласс *Набора* добавляет сообщение *добавить первым:*. При помощи этой информации программист может определить какой метод в действительности вы-

полняется при посылке сообщения *добавить первым:* набору. Следующее предложение собирает каждый такой класс во *Множество* с именем подклассы.

```
подклассы ← Множество новый.
Набор
  делать для всех подклассов: [
    :класс |
    ( класс содержит селектор: #добавить первым: )
      истина: [подклассы добавить: класс. ]. ].
```

Ту же информацию можно получить при помощи:

```
Набор
  выбрать подклассы: [:класс | класс содержит селектор: #добавить первым:. ].
```

В обоих случаях создаётся *Множество* из трёх подклассов: *Связанный список*, *Упорядоченный набор* и *Ряд серий*.

Следующее выражение возвращает набор надклассов *Малого целого* которые реализуют сообщение =.

```
Малое целое выбрать надклассы: [:класс | класс содержит селектор: #'='. ]
```

Ответ это:
Множество (Целое Величина Объект)

Несколько подклассов *Набора* реализуют сообщение *первый*. Допустим нужно посмотреть исходный текст для каждой реализации. Следующие предложения печатают исходный текст в файл с именем "методы первый".

```
| поток |
поток ← Поток файла имя файла: 'методы первый'.
Набор
  делать для всех подклассов: [
    :класс |
    ( класс содержит селектор: #первый )
      истина: [
        класс имя печатать в: поток.
        поток пс.
```

```
        ( класс исходный текст от: #первый ) печатать в: поток.
        поток пс; пс. ]. ].
поток закрыть.
```

Содержимым файла будет:

```
Набор последовательность
'первый
  ^ сам от: 1.'

Интервал
'первый
  ^ начало.'

Связанный список
'первый
  сам проверить на пустость.
  ^ первая связь.'
```

Протокол описанный в следующих разделах обычно не используется программистами, но он может быть интересен разработчикам системы. Описанные сообщения обычно используются средой программирования когда выбирается некоторый пункт меню в графическом интерфейсе.

Не смотря на то что большинство возможностей классов определено в протоколе *Поведения*, некоторые сообщения не могут быть реализованы в нём т.к. *Поведение* не предоставлят полное описание класса. В частности, *Поведение* не описывает имена переменных экземпляра и имена переменных класса, так же оно не содержит информацию о имени класса и о комментарии класса.

Описание имени класса, комментария класса и имена переменных экземпляра задаётся в *Описании класса*, подклассе *Поведения*. У *Описания класса* два подкласса, *Класс* и *Метакласс*. *Класс* описывает имена переменных класса и переменных пула. Метаклассы разделяют переменные класса и переменные пула с их единственным экземпляром. *Класс* добавляет протокол для добавления и удаления переменных класса и переменных пула, и для создания различных видов подклассов. *Метакласс* добавляет сообщение

инициализации для создания своего подкласса, т.е. сообщения для создания метакласса для нового класса.

16.2 Класс *Описание класса*

Описание класса описывает имя класса, его комментарии и имена переменных экземпляра. Это отражается в дополнительном протоколе для доступа к имени и комментарию, и к добавлению и удалению переменных экземпляра.

Протокол экземпляров *Описания класса*

доступ к описанию класса

имя
Возвращает *Цепъ* являющуюся именем получателя.

комментарий
Возвращает *Цепъ* являющуюся коментарием получателя.

комментарий: цепь
Присваивает комментарию получателя аргумент, цепь.

добавить имя переменной экземпляра: цепь
Добавляет аргумент, цепь, как переменную экземпляра получателя.

удалить имя переменной экземпляра: цепь
Удаляет аргумент, цепь, из переменных экземпляра получателя. Если такая цепь не находится, то сообщается об ошибке.

Описание класса был создан как общий надкласс для *Класса* и *Метакласса* чтобы предоставить дальнейшее структурирование описаний классов. Это помогает поддерживать общую среду разработки программ. В частности *Описание класса* добавляет структуру для организации пар селектор/метод словаря методов. Эта организация является простой категоризацией при помощи которой группируются подмножества имён словаря, точно так как методы группировались в главах этой книги. Также *Описание класса* предоставляет механизм для помещения полного описания класса во внешний поток (файл), и механизм который отслеживает все изменения произведённые в классе.

Сами классы также группируются в категории классификаций системы. Организация глав этой части книги соответсвует категориям классов системы, например, величины, числа, наборы, объекты ядра, классы ядра и поддержка ядра. Протокол для категоризации сообщений и классов включает следующие сообщения:

Протокол экземпляров *Описания класса*

организация сообщений и классов

категория

Возвращает категорию получателя в организации системы.

категория: цепь

Помещает получателя в категорию цепь, при этом удаляет его из предыдущей категории.

удалить категорию: цепь

Удаляет все сообщения в категории с именем цепь, и удаляет саму категорию.

какая категория содержит селектор: селектор

Возвращает категорию аргумента, селектор, в организации словаря методов получателя, или возвращает *пусто* если селектор не найден.

Задав категории сообщений *Описание класса* задаёт набор сообщений для копирования сообщений из одного словаря методов в другой, изменяя или оставляя имя категории. Сообщения поддерживающие копирование состоят из:

копировать: селектор из: класс

копировать: селектор из: класс классифицировать: имя категории

копировать все: ряд селекторов из: класс

копировать все: ряд селекторов из: класс классифицировать: имя категории

копировать все категории из: класс

копировать категорию: имя категории из: класс

копировать категорию: имя категории из: класс классифицировать: имя категории

Схема категорий влияет на протокол компиляции т.к. откомпилированный метод должен помещаться в некоторую категорию. Есть два сообщения: *компилировать: текст классифицировать: имя категории* и *компилировать: текст классифицировать: имя категории уведомлять: запрощик*.

Заметьте что в следующем примере *Поведение* использует протокол печати аргументов чтобы вычислить сообщение компиляции. Этими сообщениями являются:

Протокол экземпляров *Поведения*

печать

цепь переменных класса
Возвращает *Цепь* содержащую имена каждой переменной класса из объявления переменных получателя.

цепь переменных экземпляра
Возвращает *Цепь* содержащую имена каждой переменной экземпляра из объявления переменных получателя.

цепь разделяемых пулов
Возвращает *Цепь* содержащую имена каждой переменной пула из объявления переменных получателя.

Рассмотрим пример создания класса именуемого *Контрольные записи*. Этот класс должен быть таким же как *Связанный список*, за исключением того что элементы не могут удаляться. Следовательно, класс может быть создан копированием из *Связанного списока* протоколов доступа, проверок, добавления и перечисления. Предполагается что элементы *Контрольных записей* это экземпляры подкласса *Связи* поддерживающего хранение информации об аудите. Сначала создаётся класс. Предполагается что внутренняя информация о *Связанном списке* не известна, поэтому надкласс и имена переменных должны быть получены при помощи посылки сообщения *Связанному списку*.

```
Связанный список надкласс
  подкласс: 'Контрольные записи'
  имена переменных экземпляра: Связанный список цепь переменных экземпляра
```

имена переменных класса: Связанный список цепь переменных класса

словари пула: Связанный список цепь разделяемых пулов

категория: 'Record Keeping'.

Класс *Контрольные записи* заздаётся как подкласс надкласса *Связанного списка* (Связанный список надкласс). После этого копируются нужные категории из класса *Связанный список*.

Контрольные записи копировать категорию: #accessing из: Связанный список.
Контрольные записи копировать категорию: #testing из: Связанный список.
Контрольные записи копировать категорию: #adding из: Связанный список.
Контрольные записи копировать категорию: #enumerating из: Связанный список.
Контрольные записи копировать категорию: #private из: Связанный список.

Класс *Контрольные записи* объявляет две переменных экземпляра: *первая связь* и *последняя связь*, и копирует сообщения *первый*, *последний*, *размер*, *пустой*, *добавить:*, *добавить первым:* и *добавить последним:*. Также копируются все сообщения из категории собственые т.к. предполагается что хотя бы одно из этих сообщений требуется для реализации внешних сообщений.

Некоторые сообщения *Описания класса* которые поддерживают помещение описания класса во внешний поток:

Протокол экземпляров *Описания класса*

хранение в файле

вывести в файл: **поток файла**
Помещает описание получателя в файл связанный с аргументом, поток файла.

вывести в файл категорию: **имя категории**
Создаёт файл с именем равным имени получателя с расширением '.st'. Помещает в этот файл сообщения из категории *имя категории*.

вывести в файл изменённые сообщения: множество изменений в: поток файла

Аргумент, множество изменений, это набор пар класс/сообщение которые были изменены. Помещает описание каждой пары в файл доступный аргументу, поток файла.

Можно записать описание класса *Контрольные записи* в файл 'Котрольне записи.st' выполнив предложение:

Контрольные записи вывести в файл: (Поток файла имя файла: 'Котрольне записи.st'.).

16.3 Класс *Метакласс*

Главной задачей метаклассов в системе Смолток является предоставление протокола для инициализации переменных класса и для создания инициализированных экземпляров единственных экземпляров метаклассов. Поэтому главное сообщение добавляемое *Метаклассом* это само инициализирующие сообщение которое посылается *Метаклассу* чтобы создать его подкласса, и сообщение посылаемое экземпляру *Метакласса* для создания единственного экземпляра.

Протокол класса *Метакласс*

создание экземпляров

subclassOf: superMeta

Возвращает экземпляр *Метакласса* являющегося подклассом метакласса, superMeta.

name: newName environment: aSystemDictionary subclassOf: superclass instanceVariabieNames: stringOrInstVarNames variable: variableBoolean words: wordBoolean pointers: pointerBoolean classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryName comment: commentString changed: changed

Чтобы создать полностью инициализированный класс нужны все эти аргументы.

Среда программирования Смолтока предоставляет упрощённый способ, используется графический интерфейс, при помощи которого пользователь задаёт информацию для создания классов.

16.4 Класс *Класс*

Экземпляры *Класса* описывают состояние и поведение объектов. *Класс* добавляет более полную поддержку программирования возможностей по сравнению с предоставляемыми *Поведением* возможностями, и болше возможностей описания по сравнению с предоставляемыми *Описанием класса* возможностями. В частности, *Класс* добавляет представление для имён переменных класса и разделяемых переменных.

Протокол экземпляров *Класса*

доступ к экземплярам и переменным

добавить имя переменной класса: цепь
: Добавляет аргумент, цепь, в качестве переменной класса получателя. Первая буква цепи должна быть большой; цепь не может быть уже существующим именем переменной класса.

удалить имя переменной класса: цепь
: Удаляет переменную класса получателя чьё имя это аргумент, цепь. Сообщается об ошибке если это не переменная класса или если эта переменная всё ещё используется методами класса.

добавить разделяемый пул: пул
: Добавляет аргумент, пул, в качестве разделяемого пула. Сообщаетя об ошибке если этот пул уже содержится в получателе.

удалить разделяемый пул: пул
: Удаляет аргумент, пул, из разделяемых пулов получателя. Сообщается об ошибке если пул не содержится в получателе.

пул класса
: Возвращает словарь переменных класса получателя.

инициализировать
: Инициализирует переменные класса.

Дополнительные сообщения доступа помещают описание класса в файл, этот файл имеет то же имя что и имя класса (вывести в файл), и удаляют класс из системы (удалить из системы).

В словаре методов *Класса* определяются четыре вида сообщений для создания различных видов подклассов. *Класс*, также, предоставляет сообщение для переименования класса (переименовать: цепь); это сообщение предоставляется *Классом* а не *Описанием класса* т.к. метакласс нельзя переименовать.

Протокол экземпляров *Класса*

создание экземпляров

подкласс: цепь имя класса имена переменных экземпляра: цепь имена переменных экземпляра имена переменных класса: цепь имена переменных класса словари пула: цепь имён пулов категория: цепь имени категории

Создаёт новый класс фиксированной длинны (обычный) как подкласс получателя. Каждый из аргументов предоставляет информацию нужную для инициализации нового класса и его категоризации.

Три других сообщения подобных предыдущему за исключением того что первое ключевое слово это: *переменный подкласс:*, *переменный подкласс байтов:* или *переменный подкласс слов:*; поддерживают создание других видов классов. Заметьте что система требует чтобы подклассом переменной длинны тоже был класс переменной длинны. Когда возможно система производит соответствующее преобразование; иначе программисту сообщается об ошибке.

Допустим что каждый раз при создании подкласса, нужно создавать сообщения для присваивания и получения переменных экземпляра этого класса. Например, если создан класс *Запись* с переменными экземпляра *имя* и *адрес*, нужно предоставить сообщения и именами *имя* и *адрес*, чтобы возвращать эти переменные, и *имя: аргумент* и *адрес: аргумент* для присваивания значений этим переменным значения аргумента сообщения. Один из способов достичь

этого — добавить следующий метод в протокол создания экземпляров класса *Класс*.

```
подкласс доступа: имя класса
имена переменных экземпляра: цепь переменных экземпляра
имена переменных класса: цепь переменных класса
словари пула: цепь имён пулов
категория: имя категории
  | новый класс |
  новый класс ← сам
    подкласс: имя класса
    имена переменных экземпляра: цепь переменных экземпляра
    имена переменных класса: цепь переменных класса
    словари пула: цепь имён пулов
    категория: имя категории.
  новый класс имена переменных экземпляра
    делать: [
      :имя |
      новый класс
        компилировать: имя, ' ↑', имя, '.'
        классифицировать: #accessing.
      новый класс
          компилировать: имя, ': аргумент ', имя, ' ← аргумент.
↑аргумент.'
          классифицировать: #accessing.].
  ↑ новый класс.
```

Этот метод создаёт класс обычным образом, затем для каждого имени переменной экземпляра компилируются два метода. Первый вида:

```
имя
  ↑ имя.
```

и второй вида:

```
имя: аргумент
  имя ← аргумент.
  ↑ аргумент.
```

Поэтому если нужно создать класс *Запись*, то можно его создать послав *Объекту* следующее сообщение:

```
Объект
  подкласс доступа: 'Запись'
  имена переменных экземпляра: 'имя адрес'
  имена переменных класса: ''
  словари пула: ''
  категория: 'Example'.
```

Сообщение находится в словаре методов *Класса*, и создаёт следующие четыре сообщения в категории accessing класса *Запись*.

```
имя класса Запись

accessing

имя
  ↑имя.
имя: аргумент
  имя ← аргумент.
  ↑аргумент.
адрес
  ↑адрес.
адрес: аргумент
  адрес ← аргумент.
  ↑аргумент.
```

Глава 17

TheProgrammingInterface

Глава 18

Graphics Kernel

Глава 19

Pens

Глава 20

Display Objects

Часть III

Пример разработки и реализации небольшого приложения

Глава 21

Probability Distributions

Глава 22

Event-Driven Simulations

Глава 23

Statistics Gathering

Глава 24

The Use of Resources

Глава 25

Coordinated Resources

Часть IV

Определение вирутальной машины Смолтока

Предыдущие три части этой книги описывали систему Смолток с точки зрения программиста. Пять глав этой части представляют систему с точки зрения реализации. Читатели не интересующиеся реализацией системы могут пропусить эти главы. Читатель интересующиеся общими чертами реализации могут прочитать только главу 26. Читатели интересующиеся деталями реализации, в том числе как в действиетльности реализована система, должны также прочитать оставшиеся четыре главы.

Глава 26

Реализация

Оглавление

Можно выделить две главных части системы Смолток: виртуальный образ и виртуальную машину.

1. Виртуальный образ содержит все объекты системы.

2. Виртуальная машина состоит из устройств и функций на машинном языке (или микрокоде), которые придают движение объектам виртуального образа.

Задача реализующего систему — создать виртуальную машину. Затем виртуальный образ может быть загружен в эту виртуальную машину и система Смолток станет интерактивной сущностью описанной в предыдущих главах.

Обзор реализации Смолтока данный в этой главе организован в виде обзора сверху вниз, начиная от исходных методов написанных программистом. Эти методы переводятся компилятором в последовательность инструкций называемых байткодами. Компилятор и байткоды это тема первого раздела этой главы. Байткоды созданные компилятором это инструкции для интерпретатора, они описываются в следующем разделе. За интерпретатором следует реализация памяти объектом которая хранит объекты составляющие виртуальный образ. Память объектов описывается в третьем разделе этой главы. В самом низу любой реализации находится оборудование. Четвёртый и последний разделы этой главы обсуждают оборудование требуемое для реализации интерпретатора и памяти объектов. Главы 27-30 дают детальное определение интерпретатора виртуальной машины и памяти объектов.

26.1 Компилятор

Исходные методы пишущиеся программистами представляются системой Смолток как экземпляры *Цепи*. Эти цепи содержат последовательности знаков которые удовлетворяют синтаксису введённому в первой части этой книги. Например следующий исходный метод может описывать как экземпляры класса *Прямоугольник* отвечают на унарное сообщение *центр*. Сообщение *центр* используется для нахождения *Точки* равноудалённой от четырёх сторон прямоугольника.

```
центр
  ↑ начало + угол / 2.
```

Исходные методы переводятся компилятором системы в последовательность инструкций стэк-ориентированного интерпретатора. Инструкции это восьмибитные числа называемые байткодами. Например байткоды соответствующие исходному методу показанному выше это:

0, 1, 176, 119, 185, 124

Т.к. значения байткодов мало говорят об их значении для интерпретатора, в этой главе будут даваться списки байткодов с комментарием об их функции. Любая часть комментария байткода которая

зависит от контекста метода в котором он встречается будет заключаться в скобки. Часть комментария без скобок описывает общую функцию байткода. Например, байткод 0 всегда указывает интерпретатору поместить значение первой переменной экземпляра получателя на стэк. То что это переменная именуется *начало* зависит от того что этот метод используется *Прямоугольником*, поэтому *начало* заключено в скобки. Ниже показан метод *Прямоугольника центр* с комментариями.

***Прямоугольник* центр**

0	поместить значение первой переменной экземпляра получателя (начало) на стэк
1	поместить на стэк значение второй переменной экземпляра получателя (угол)
176	послать бинарное сообщение с селектором +
119	поместить на стэк *Малое целое* 2
185	послать бинарное сообщение с селектором /
124	возвратить объект с вершины стэка как значение сообщения (центр)

Стэк, упоминаемый в некоторых байткодах используется для нескольких целей. В этом методе он используется для помещения получателя, аргументов и результатов двух посылаемых сообщений. Также стэк используется в качестве источника результата возвращаемого из метода *центр*. Стэк управляется интерпретатором и будет описан более подробно в следующем разделе. Описание всех типов байткодов будет приведено в конце этого раздела.

Программист не взаимодействует с компилятором напрямую. При добавлении нового исходного метода в класс (*Прямоугольник* в данном примере), класс запрашивает у компилятора экземпляр *Откомпилированного метода* содержащий перевод исходного метода в байткоды. Класс предоставляет компилятору некоторую необходимую информацию не присутствующую в исходном методе, в неё входят имена переменных экземпляра и словари содержащие доступные разделяемые переменные (глобальные, класса и переменные пулов). Компилятор переводит исходный текст в *Откомпилированный метод* и класс помещает его в свой словарь сообщений.

Например, *Откомпилированный метод*, показанный выше, помещается в словарь сообщений *Прямоугольника* по ключу центр.

Другой пример байткодов скомпилированных из исходного метода показывает использование байткодов помещения. Сообщение *Прямоугольника размеры:* изменяет ширину и высоту получателя так чтобы они стали равны координате аргумента икс и игрек. Левый верхний угол получателя (начало) не изменяется, а правый нижний угол (угол) сдвигается.

```
размеры: новые размеры
  угол ← начало + новые размеры.
```

***Прямоугольник* размеры:**

0	поместить значение первой переменной экземпляра получателя (начало) на стэк
16	поместить на стэк первую временную переменную (новые размеры)
176	послать бинарное сообщение с селектором +
97	снять со стэка верхний объект и поместить его во вторую переменную экземпляра получателя (угол)
120	возвратить получателя как значение сообщения (размеры:)

Форма исходных методов и откомпилированных байткодов отличаются в нескольких моментах. Имена переменных из исходных методов переводятся в инструкции помещения объектов на стэк, селекторы переводятся в инструкции посылки сообщений и стрелка вверх переводится в инструкцию возвращения результата. Порядок соответствующих частей также отличается в исходных методах и в откомпилированных байткодах. Не смотря на эти отличия в форме, исходный метод и откомпилированные байткоды описывают те же действия.

26.1.1 Откомпилированные методы

Компилятор создаёт экземпляры *Откомпилированного метода* для хранения перевода в байткоды исходного метода. В допол-

нение к самим байткодам, *Откомпилированный метод* содержит набор объектов называемый его *блоком литералов*. Блок литералов содержит любые объекты которые не могут быть представлены напрямую байткодами. Ссылки на все объекты в сообщениях *Прямоугольника центр* и *размеры:* делаются напрямую байткодами, поэтому *Откомпилированные методы* для этих методов не требуют блока литералов. В качестве примера *Откомпилированного метода* с блоком литералов рассмотрим метод *Прямоугольника пересекает:*. Сообщение *пересекает:* определяет перекрывает ли один *Прямоугольник* (получатель) другой *Прямоугольник* (аргумент).

пересекает: прямоугольник
↑ (начало макс: прямоугольник начало) < (угол мин: прямоугольник угол).

Четыре селектора сообщений: *макс:*, *начало*, *мин:* и *угол* не входят в набор объектов на которые можно напрямую сослаться байткодами. Эти селекторы включены в блок литералов *Откомпилированного метода* и байткоды посылки ссылаются на эти селекторы по их положению в блоке литералов. Блок литералов *Откомпилированного метода* будет показываться после байткодов.

***Прямоугольник* пересекает:**

0	поместить значение первой переменной экземпляра получателя (начало) на стэк
16	поместить на стэк первую временную переменную (прямоугольник)
209	послать унарное сообщение со вторым селектором из блока литералов (начало)
224	послать одноаргументное сообщение с первым селектором из блока литералов (макс:)
1	поместить на стэк значение второй переменной экземпляра получателя (угол)
16	поместить на стэк первую временную переменную (прямоугольник)
211	послать унарное сообщение с четвёртым селектором из блока литералов (угол)

226	послать унарное сообщение с третьим селектором из блока литералов (мин:)
178	послать бинарное сообщение с селектором <
124	возвратить объект с вершины стэка как значение сообщения (пересекает:)

блок литералов
#макс:
#начало
#мин:
#угол

Категории объектов на которые можно напрямую сослаться байт-кодами:

- получатель и аргументы выполняемого сообщения
- значения переменных экземпляра получателя
- значения любых временных переменных, требуемых методу
- семь специальных констант (истина, ложь, пусто, -1, 0, 1 и 2)
- 32 специальных селекторов сообщений

Ниже показаны 32 специальных селектора сообщений.

+	−	<	>
<=	>=	=	~ =
*	/	\	@
сдвинуть биты: (от:) (пом следующим:)	\\ (от:пом:) (в конце)	побитовое и: (размер) ==	побитовое или: (следующий) класс
экземпляр блока: (новый)	значение (новый:)	значение: (икс)	(делать:) (игрек)

Селекторы в скобках могут быть заменены другими селекторами при помощи изменения компилятора и перекомпилирования всех методов системы. Другие селекторы встроены в виртуальную машину.

Любой объект, на который ссылаются байткоды *Откомпилированного метода*, который не входит ни в одну из пяти вышеуказанных категорий должен быть помещён в блок литералов. Обычно в блоке литералов содержатся:

- разделяемые переменные (глобальные, класса и пула)
- большинство констант литералов (числа, знаки, цепи, ряды и символы)
- большинство селекторов сообщений (те что не являются специальными)

Объекты этих трёх типов могут быть перемешаны в блоке литералов. Если на объект в блоке литералов есть две ссылки из одного метода, то нужно только одно вхождение объекта в таблицу литералов. Два байткода ссылающихся на этот объект будут ссылаться на ту же позицию в блоке литералов.

Два типа объектов указанных выше, временные переменные и разделяемые переменные, не будут использоваться в примерах методов. Следующий пример метода *Прямоугольника объединить:* использует оба типа. Сообщение *объединить:* используется для нахождения *Прямоугольника* который содержит площади и получателя и аргумента.

```
объединить: прямоугольник
  | точка мин точка макс |
  точка мин ← начало мин: прямоугольник начало.
  точка макс ← угол макс: прямоугольник угол.
  ↑ Прямоугольник начало: точка мин угол: точка макс.
```

Когда *Откомпилированный метод* использует временные переменные (*точка макс* и *точка мин*) требуемое их число задаётся в первой строке описания байткодов. Когда *Откомпилированный метод* использует разделяемые переменные (*Прямоугольник*

в этом примере) в блок литералов помещается экземпляр *Ассоциации*. Все *Откомпилированные методы* которые ссылаются на данную разделяемую переменную содержат в блоке литералов одну и ту же *Ассоциацию*.

***Прямоугольник* объединить:** требует 2 временные переменные

0	поместить значение первой переменной экземпляра получателя (начало) на стэк
16	поместить на стэк первую временную переменную (аргумент прямоугольник)
209	послать унарное сообщение со вторым селектором из блока литералов (начало)
224	послать одноаргументное сообщение с первым селектором из блока литералов (мин:)
105	снять со стэка верхний объект и поместить его во вторую временную переменную (точка мин)
1	поместить на стэк значение второй переменной экземпляра получателя (угол)
16	поместить на стэк первую временную переменную (аргумент прямоугольник)
211	послать унарное сообщение с четвёртым селектором из блока литералов (угол)
226	послать унарное сообщение с третьим селектором из блока литералов (макс:)
106	снять со стэка верхний объект и поместить его в третью временную переменную (точка макс)
69	поместить на стэк значение разделяемой переменной из шестой позиции в блоке литералов (Прямоугольник)
17	поместить на стэк вторую временную переменную (точка мин)
18	поместить на стэк третью временную переменную (точка макс)
244	послать сообщение с двумя аргументами и пятым селектором из блока литералов (начало:угол:)
124	возвратить объект с вершины стэка как значение сообщения (объединить:)

блок литералов
#мин:
#начало
#макс:
#угол
#начало:угол:
Ассоциация: #Прямоугольник → Прямоугольник

Временные переменные

Временные переменные создаются для конкретного выполнения *Откомпилированного метода* и перестают существовать при завершении выполнения метода. *Откомпилированный метод* указывает интерпретатору количество требуемых ему временных переменных. Аргументы выполняемого сообщения и значения временных переменных вместе помещаются в область временных переменных. Сначала помещаются аргументы а затем временные переменные. Доступ к ним осуществляется одним типом байткодов (чей комментарий ссылается на временные переменные). Т.к. сообщение *объединить:* использует один аргумент, то его временные переменные используют вторую и третью позицию в области временных переменных.

Разделяемые переменные

Разделяемые переменные находятся в словарях.

- глобальные переменные в словаре чьи имена могут быть доступны любому методу
- переменные класса в словаре чьи имена могут быть доступны только методам одного класса и его подклассам
- переменные пула в словаре чьи имена могут быть доступны методам нескольких классов

Разделяемые переменные это ассоциации которые составляют эти словари. Система представляет ассоциации в общем, и разделяемые переменные в частности, как экземпляры *Ассоциации*. Когда компилятор встречает в исходном методе имя разделяемой переменной в область литералов *Откомпилированного метода* включается *Ассоциация* с тем же именем. Байткод который обращается к разделяемой переменной указывает положение *Ассоциации* в области литералов. Действительное значение переменной храниться в переменной экземпляра *Ассоциации*. *Откомпилированный метод* для сообщения *объединить:*, показанного выше, ссылается на класс *Прямоугольник* при помощи включения из глобального словаря *Ассоциации* с именем являющимся символом *#Прямоугольник* и значением являющимся классом *Прямоугольник*.

26.1.2 Байткоды

Интерпретатор понимает 256 байткодов которые распределены на пять категорий: помещение, сохранение, посылка, возврата и прыжки. Этот раздел даёт общее описание каждого типа байткодов без детального объяснения какой байткод какую инструкцию представляет. Глава 28 описывает точный смысл каждого байткода. Т.к. интерпретатору требуется более чем 256 инструкций, то некоторые из байткодов имеют расширение. Расширение это один или два байта следующие за байткодом, они уточняют инструкцию. Расширение это не инструкция, это только часть инструкции.

Байткоды помещения

Байткоды помещения указывают исходные объекты которые нужно поместить на вершину стэка интерпретатора. Источники объектов это:

- получатель сообщения, выполняющий *Откомпилированный метод*
- переменные экземпляра получателя
- область временных переменных (аргументы сообщения и временные переменные)

- область литералов *Откомпилированного метода*
- вершина стэка (т.е. этот байткод удваивает вершину стэка)

Примеры большинства типов байткодов помещения использовались в примерах. Байткод который удваивает вершину стэка используется для реализации сообщений каскадов.

Два других типа байткодов используют блок литералов в качестве источников объектов. Один тип используется для помещения литералов констант, а другой тип используется для помещения значений разделяемых переменных. Константы литералы напрямую помещаются в блок литералов, но значения разделяемых переменных помещаются в *Ассоциации* которые хранятся в блоке литералов. Следующий пример метода использует одну разделяемую переменную и одну константу литерал.

увеличить номер
↑Номер ← Номер + 4.

***Класс пример* увеличить номер**

64	поместить на стэк разделяемую переменную из первой позиции блока литералов (Номер)
33	поместить на стэк константу из второй позиции блока литералов (4)
176	послать бинарное сообщение с селектором +
129, 192	поместить объект с вершины стэка в разделяемую переменную в первой позиции блока литералов (Номер)
124	возвратить объект с вершины стэка как значение сообщения (увеличить номер)

блок литералов
Ассоциация: #Номер → 260
4

Байткоды сохранения

Байткоды компилируемые из выражения присваивания заканчиваются на байткод сохранения. Байткоды перед байткодом сохранения вычисляют новое значение переменной и оставляют его на вершине стэка. Байткод сохранения указывает переменную чьё значение нужно изменить. Переменные которые можно изменять:

- переменные экземпляра получателя
- временные переменные
- разделяемые переменные

Некоторые байткоды сохранения удаляют сохраняемый объект со стэка, а некоторые после сохранения оставляют объект на вершине стэка.

Байткоды посылки

Байткод посылки задаёт селектор сообщения и количество аргументов требуемых сообщению. Получатель и аргументы сообщения берутся интерпретатором со стэка, получатель находится под аргументами. При встрече байткода выполняется посылка сообщения, результат сообщения замещает получателя и аргументы на вершине стэка. Детали посылки сообщений и возвращаемых значений это тема следующих разделов этой главы. Набор из 32-х байткодов посылки напрямую ссылаются на специальные селекторы указанные выше. Другие байткоды посылки ссылаются на свои селекторы в блок литералов.

Байткоды возврата

При встрече байткода возвращения выполнение *Откомпилированного метода*, в котором он находится, полностью завершается. Поэтому значение возвращается сообщению которое вызвало этот *Откомпилированный метод*. Значение обычно помещается на вершину стэка. Четыре специальных байткода возвращения возвращают получателя сообщения (*себя*), *истину*, *ложъ* и *пусто*.

Байткоды прыжков

Обычно интерпретатор выполняет байткоды последовательно в порядке их появления в *Откомпилированном методе*. Байткоды прыжков указывают что следующий выполняемый байткод это не следующий байткод. Есть два варианта прыжков: безусловные и условные. Безусловные прыжки передают управление когда они встречаются. Условные прыжки передают управление только если вершиной стэка является заданное значение. Некоторые условные прыжки передают управление если верхний объект стэка это *истина*, а другие если это *ложь*. Байткоды прыжков используются для эффективной реализации управляющих конструкций.

Оптимизированные так компилятором управляющие конструкции это сообщения условного выбора *Логики* (*истина:*, *ложь:* и *истина:ложь:*), некоторые сообщения логических операций *Логики* (*и:* и *или:*) и сообщения условного повторения блоков (*пока истина:* и *пока ложь:*). Байткоды прыжков указывают следующий исполняемый байткод относительно положения прыжка. Другими словами, они говорят интерпретатору сколько пропустить байткодов. Следующий метод *Прямоугольника содержит точку:* использует условный прыжок.

```
содержит точку: точка
  начало <= точка истина: [↑точка < угол.] ложь: [↑ложь.].
```

***Прямоугольник* содержит точку:**

0	поместить значение первой переменной экземпляра получателя (начало) на стэк
16	поместить на стэк первую временную переменную (аргумент точка)
180	послать бинарное сообщение с селектором <=
155	прыгнуть вперёд через 4 байткода если объект на вершине стэка это ложь
16	поместить на стэк первую временную переменную (аргумент точка)
1	поместить на стэк значение второй переменной экземпляра получателя (угол)
178	послать бинарное сообщение с селектором <

124 возвратить объект с вершины стэка как значение сообщения (содержит точку:)

122 возвратить ложь как значение сообщения (содержит точку:)

26.2 Интерпретатор

Интерпретатор Смолтока выполняет инструкции байткодов находящиеся в *Откомпилированном методе*. Интерпретатор использует пять видов информации и постоянно выполняет три шага цикла.

Состояние интерпретатора

1. *Откомпилированный метод*, чьи байткоды выполняются.
2. Положение следующего байткода *Откомпилированного метода* который будет выполнен. Это указатель инструкции интерпретатора.
3. Получатель и аргументы сообщения которое выполняет *Откомпилированный метод*.
4. Временные переменные нужные *Откомпилированному методу*.
5. Стэк.

Выполнение большинства байткодов использует стэк интерпретатора. Байткоды помещения указывают где найти объекты помещаемые на стэк. Байткоды сохранения указывают куда поместить объект находящейся на стэке. Байткоды посылки удаляют получателя и аргументы сообщения со стэка. Когда вычисляется результат сообщения, он помещается на стэк.

Цикл интерпретатора

1. Извлечь из *Откомпилированного метода* байткод на который установлен указатель инструкции.
2. Увеличить указатель инструкции.

3. Выполнить функцию заданную байткодом.

В качестве примера работы интерпретатора проследим выполнение *Откомпилированного метода* для метода *центр Прямоугольника*. Состояние интерпретатора будет указываться после его каждого цикла. Указатель инструкции будет показываться при помощи стрелки указывающей на следующий выполняемый байткод *Откомпилированного метода*.

⇒	**0**	поместить значение первой переменной экземпляра получателя (начало) на стэк

Получатель, аргументы, временные переменные и объекты стэка будут показываться в их печатном виде (ответ на сообщение *цепь для печати*). Например, если сообщение посылается *Прямоугольнику*, то получатель будет показан так:

Получатель 100@100 угол: 200@200

В начале выполнения стэк пуст и указатель инструкции показывает на первый байткод *Откомпилированного метода*. Этот *Откомпилированный метод* не требует временных переменных и вызываемые сообщения не имеют аргументов, поэтому эти две категории пусты.

Метод *Прямоугольника* центр

⇒	**0**	поместить значение первой переменной экземпляра получателя (начало) на стэк
	1	поместить на стэк значение второй переменной экземпляра получателя (угол)
	176	послать бинарное сообщение с селектором +
	119	поместить на стэк *Малое целое* 2
	185	послать бинарное сообщение с селектором /
	124	возвратить объект с вершины стэка как значение сообщения (центр)

Получатель 100@100 угол: 200@200

Аргументы
Временные переменные
Стэк

Через один цикл интерпретатора, указатель инструкции будет продвинут и значение первой переменной получателя скопируется на стэк.

Метод *Прямоугольника* центр

	0	поместить значение первой переменной экземпляра получателя (начало) на стэк
⇒	**1**	поместить на стэк значение второй переменной экземпляра получателя (угол)
	176	послать бинарное сообщение с селектором +
	119	поместить на стэк *Малое целое* 2
	185	послать бинарное сообщение с селектором /
	124	возвратить объект с вершины стэка как значение сообщения (центр)

Получатель	100@100 угол: 200@200
Аргументы	
Временные переменные	
Стэк	100@100

Результат второго цикла интерпретатора подобен первому. Вершина стэка показана к низу страницы. Это соответствует общеиспользуемому соглашению что положения в памяти показываются с адресами растущими к низу страницы.

Метод *Прямоугольника* центр

	0	поместить значение первой переменной экземпляра получателя (начало) на стэк
	1	поместить на стэк значение второй переменной экземпляра получателя (угол)
⇒	**176**	послать бинарное сообщение с селектором +
	119	поместить на стэк *Малое целое* 2
	185	послать бинарное сообщение с селектором /

124 возвратить объект с вершины стэка как значение сообщения (центр)

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	100 @ 100 200 @ 200

На третьем цикле интерпретатор встречает байткод посылки. Он удаляет со стэка два объекта и использует их в качестве получателя и аргумента сообщения с селектором +. Процедура посылки сообщения не будет детально рассматриваться здесь. Сейчас нужно знать что в конце концов результат сообщения + будет помещён на стэк. Посылка сообщения будет описана в следующих главах.

Метод *Прямоугольника* центр

0 поместить значение первой переменной экземпляра получателя (начало) на стэк
1 поместить на стэк значение второй переменной экземпляра получателя (угол)
176 послать бинарное сообщение с селектором +
⇒ **119** поместить на стэк *Малое целое* 2
185 послать бинарное сообщение с селектором /
124 возвратить объект с вершины стэка как значение сообщения (центр)

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	300 @ 300

Следующий цикл интерпретатора помещает на стэк константу 2.

Метод *Прямоугольника* центр

0 поместить значение первой переменной экземпляра получателя (начало) на стэк

	1	поместить на стэк значение второй переменной экземпляра получателя (угол)
	176	послать бинарное сообщение с селектором +
	119	поместить на стэк *Малое целое* 2
⇒	**185**	послать бинарное сообщение с селектором /
	124	возвратить объект с вершины стэка как значение сообщения (центр)

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	300 @ 300
	2

Следующий цикл интерпретатора посылает другое сообщение чей результат заменяет на стэке его получателя и аргумент.

Метод *Прямоугольника* центр

	0	поместить значение первой переменной экземпляра получателя (начало) на стэк
	1	поместить на стэк значение второй переменной экземпляра получателя (угол)
	176	послать бинарное сообщение с селектором +
	119	поместить на стэк *Малое целое* 2
	185	послать бинарное сообщение с селектором /
⇒	**124**	возвратить объект с вершины стэка как значение сообщения (центр)

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	150 @ 150

Последний байткод возвращает результат сообщения *центр*. Результат находится на стэке (150 @ 150). Здесь видно что байткод возвращения должен вызвать помещение результата на другой стэк. Детали возвращения значения сообщения будут описаны после опи-

сания посылки сообщения.

26.2.1 Контексты

Байткоды помещения, сохранения и прыжков требуют только маленьких изменений *Контекста* интерпретатора. Объекты могут быть перемещены на или со стэка, и всегда изменяется указатель инструкции; но большая часть состояния остаётся неизменной. Байткоды посылки и возвращения могут потребовать больших изменений состояния интерпретатора. При посылке сообщения, чтобы выполнить различные *Откомпилированные методы*, могут измениться все пять частей состояния интерпретатора. Старое состояние интерпретатора должно запоминаться т.к. после посылки сообщения и возвращения значения байткоды должны продолжать выполняться.

Интерпретатор сохраняет своё состояние в объектах называемых контекстами. Одновременно в системе существуют много контекстов. Контекст представляющий текущее состояние интерпретатора называется *активным контекстом*. Когда байткод в активном контексте *Откомпилированного метода* требует для выполнения нового *Откомпилированного метода* активный контекст становится приостановленным и создаётся новый контекст и становится активным. Приостановленный контекст хранит состояние связанное с исходным *Откомпилированным методом* до тех пока он снова не станет активным. Контекст должен помнить контекст который он сделал приостановленным чтобы возобновить его при возвращении значения. Приостановленный контекст называется новым *отправителем* контекста.

Форма показа состояния интерпретатора используемая в последнем разделе будет также использоваться для контекстов. Активный контекст будет указываться словом **Активный** в своём верхнем разделителе. Приостановленные контексты будут указывать **Пассивный**. Рассмотрим, например, контекст представляющий выполнение *Откомпилированного метода Прямоугольника* для селектора *правый центр* с получателем 100 @ 100 угол: 200 @ 200. Исходный метод для него:

правый центр

↑сам правый @ сам центр игрек.

Ниже показано состояние интерпретатора после выполнения первого байткода. Отправитель это некоторый другой контекст системы.

Активный

Метод *Прямоугольника* правый центр

	112	поместить на стэк получателя (себя)
⇒	**208**	послать унарное сообщение с первым селектором из блока литералов (правый)
	112	поместить на стэк получателя (себя)
	209	послать унарное сообщение со вторым селектором из блока литералов (центр)
	207	послать унарное сообщение с селектором игрек
	187	послать унарное сообщение с селектором @
	124	возвратить объект с вершины стэка как значение сообщения (правый центр)

блок литералов
#правый
#центр

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	100 @ 100 угол: 200 @ 200
Отправитель ⇓	

После выполнения следующего байткода этот контекст становится приостановленным. Объект помещённый первым байткодом удаляется для использования в качестве получателя нового контекста, который становится активным. Новый активный контекст показан над приостановленным контекстом.

Активный

Метод *Прямоугольника* правый

⇒	**1**	поместить на стэк значение второй переменной экземпляра получателя (угол)
	206	послать унарное сообщение с селектором икс
	124	возвратить объект с вершины стэка как значение сообщения (правый)

Получатель	100@100 угол: 200@200
Аргументы	
Временные переменные	
Стэк	
Отправитель ⇓	

Пассивный

Метод *Прямоугольника* правый центр

	112	поместить на стэк получателя (себя)
	208	послать унарное сообщение с первым селектором из блока литералов (правый)
⇒	**112**	поместить на стэк получателя (себя)
	209	послать унарное сообщение со вторым селектором из блока литералов (центр)
	207	послать унарное сообщение с селектором игрек
	187	послать унарное сообщение с селектором @
	124	возвратить объект с вершины стэка как значение сообщения (правый центр)

блок литералов
#правый
#центр

Получатель	100@100 угол: 200@200
Аргументы	
Временные переменные	
Стэк	
Отправитель ⇓	

На следующий цикле интерпретатор продвигает новый контекст

вместо предыдущего.

Активный

Метод *Прямоугольника* правый

1	поместить на стэк значение второй переменной экземпляра получателя (угол)
⇒ **206**	послать унарное сообщение с селектором икс
124	возвратить объект с вершины стэка как значение сообщения (правый)

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	
Стэк	200 @ 200
Отправитель ⇓	

Пассивный

Метод *Прямоугольника* правый центр

112	поместить на стэк получателя (себя)
208	послать унарное сообщение с первым селектором из блока литералов (правый)
⇒ **112**	поместить на стэк получателя (себя)
209	послать унарное сообщение со вторым селектором из блока литералов (центр)
207	послать унарное сообщение с селектором игрек
187	послать унарное сообщение с селектором @
124	возвратить объект с вершины стэка как значение сообщения (правый центр)

блок литералов
#правый
#центр

Получатель	100 @ 100 угол: 200 @ 200
Аргументы	
Временные переменные	

Стэк
Отправитель ⇓

На следующем цикле посылается другое сообщение, возможно создающее другой контекст. Вместо описания ответа на сообщение икс мы пропустим этот процесс до шага на котором контекст возвращает значение (контексту правый). Когда будет получен результат от икса контекст будет выглядеть так:

Активный

Метод *Прямоугольника* правый

1	поместить на стэк значение второй переменной экземпляра получателя (угол)
206	послать унарное сообщение с селектором икс
⇒ **124**	возвратить объект с вершины стэка как значение сообщения (правый)

Получатель	100@100 угол: 200@200
Аргументы	
Временные переменные	
Стэк	200
Отправитель ⇓	

Пассивный

Метод *Прямоугольника* правый центр

112	поместить на стэк получателя (себя)
208	послать унарное сообщение с первым селектором из блока литералов (правый)
⇒ **112**	поместить на стэк получателя (себя)
209	послать унарное сообщение со вторым селектором из блока литералов (центр)
207	послать унарное сообщение с селектором игрек
187	послать унарное сообщение с селектором @
124	возвратить объект с вершины стэка как значение сообщения (правый центр)

блок литералов
#правый
#центр

Получатель 100 @ 100 угол: 200 @ 200
Аргументы
Временные переменные
Стэк
Отправитель ⇓

Следующий байткод возвращает значение с вершины стэка активного контекста (200) в качестве значения сообщения которое создало контекст (правый). Отправитель активного контекста снова становится активным контекстом и возвращённое значение помещается на его стэк.

Активный

Метод *Прямоугольника* правый центр

	112	поместить на стэк получателя (себя)
	208	послать унарное сообщение с первым селектором из блока литералов (правый)
⇒	**112**	поместить на стэк получателя (себя)
	209	послать унарное сообщение со вторым селектором из блока литералов (центр)
	207	послать унарное сообщение с селектором игрек
	187	послать унарное сообщение с селектором @
	124	возвратить объект с вершины стэка как значение сообщения (правый центр)

блок литералов
#правый
#центр

Получатель 100 @ 100 угол: 200 @ 200
Аргументы
Временные переменные

Стэк 200
Отправитель ⇓

26.2.2 Контекст блока

Контексты, показанные в последнем разделе, представляются в системе при помощи экземпляров *Контекста метода*. *Контекст метода* представляет выполнение *Откомпилированного метода* в ответ на сообщение. В системе существует другой тип контекста который представляется экземплярами *Контекста блока*. *Контекст блока* представляет блок исходного метода который не является частью оптимизированной управляющей структуры. Компиляция оптимизированных управляющих структур была описана в предыдущих разделах о байткодах прыжков. Байткоды скомпилированные из неоптимизированной управляющей структуры иллюстрируются следующим гипотетическим методом *Набора*. Это метод возвращает набор классов элементов получателя.

```
классы
  ↑сам собрать: [:элемент | элемент класс.].
```

Набор **классы** требует 1 временную переменную

112	поместить на стэк получателя (себя)
137	поместить на стэк активный контекст (этот контекст)
118	поместить на стэк *Малое целое* 1
200	послать одноаргументное сообщение с селектором экземпляр блока:
164, 4	прыгнуть вперёд через 4 байта
104	снять со стэка верхний объект и поместить его в первую временную переменную (элемент)
16	поместить на стэк первую временную переменную (элемент)
199	послать унарное сообщение с селектором класс
125	возвратить объект с вершины стэка как значение блока

224	послать одноаргументное сообщение с первым селектором из блока литералов (собрать:)
124	возвратить объект с вершины стэка как значение сообщения (классы)

блок литералов
#собрать:

Контекст блока создаётся при помощи сообщения *экземпляр блока:* посылаемого активному контексту. Байткод помещающий на стэк активный контекст не был описан вместе с остальными байткодами помещения т.к. в то время не были описаны функции контекстов. Аргумент *экземпляра блока:* (1 в этом примере) указывает количество аргументов требуемых блоку. *Контекст блока* разделяет большую часть состояния с создавшим его активным контекстом. Получатель, аргументы, временные переменные, *Откомпилированный метод* и отправитель все те же самые. *Контекст блока* имеет свой собственный указатель инструкции и стэк. После возвращения из сообщения *экземпляр блока:* вновь созданный *Контекст блока* находится на стэке активного контекста и следующая инструкция перепрыгивает через байткоды описывающие действия блока. Активный контекст передаёт *Контексту блока* начальный указатель инструкции указывающий на байткод после этого байткода прыжка. После сообщения *экземпляр блока:* компилятор всегда использует расширенный (двухбайтный) прыжок, поэтому начальный указатель инструкции *Контекста блока* всегда на двойку больше указателя инструкции активного контекста при получении им сообщения *экземпляр блока:*.

Метод сообщения *Набора классы* создаёт *Контекст блока*, но не выполняет его байткоды. Когда набор получает сообщение *собрать:* он начинает повторно посылать *Контексту блока* сообщение *значение:* с элементами набора в качестве аргумента. *Контекст блока* отвечает на *значение:* становясь активным контекстом, чьи байткоды начинает выполнять интерпретатор. Перед тем как *Контекст блока* станет активным аргумент сообщения *значение:* помещается на стэк *Контекста блока*. Первый байткод выполняемый *Контекстом блока* сохраняет это значение во времен-

ную переменную используемую для аргумента блока.

Контекст блока может возвращать значение двумя способами. После выполнения байткодов блока конечное значение стэка возвращается в качестве значения сообщения *значение* или *значение:*. Также блок может вернуть значение сообщению которое вызвало *Откомпилированный метод* создавший *Контекст блока*. Это делается при помощи обычного байткода возврата. Гипотетический метод *Набора содержит экземпляр:* использует оба типа возврата из *Контекста блока*.

```
содержит экземпляр: класс
  сам
    делать: [:элемент | ( элемент это разновидность: класс ) истина:
[↑истина. ]. ].
  ↑ложь.
```

***Набор* содержит экземпляр:** требует 1 временную переменную

112	поместить на стэк получателя (себя)
137	поместить на стэк активный контекст (этот контекст)
118	поместить на стэк *Малое целое* 1
200	послать одноаргументное сообщение с селектором экземпляр блока:
164, 8	прыгнуть вперёд через 8 байтов
105	снять со стэка верхний объект и поместить его во вторую временную переменную (элемент)
17	поместить на стэк вторую временную переменную (элемент)
16	поместить на стэк первую временную переменную (класс)
224	послать одноаргументное сообщение с первым селектором из блока литералов (это разновидность:)
152	снять со стэка объект и прыгнуть через 1 байткод если это *ложъ*
121	возвратить истину как значение сообщения (содержит экземпляр:)
115	поместить на стэк *пусто*

125	возвратить объект с вершины стэка как значение блока
203	послать одноаргументное сообщение с селектором делать:
135	снять со стэка объект
122	возвратить ложь как значение сообщения (содержит экземпляр:)

блок литералов
#это разновидность:

26.2.3 Сообщения

Когда встречается байткод посылки интерпретатор находит *Откомпилированный метод* указанный в сообщении следующим образом:

1. *Находится получатель сообщения.* Получатель находится на стэке под аргументами. Количество аргументов указывается в байткоде посылки.

2. *Находится словарь сообщений.* Он находится в классе получателя.

3. *В словаре находится селектор сообщения.* Селектор указывается байткодом посылки.

4. *Если селектор найден*, то связанный с ним *Откомпилированный метод* описывает ответ на сообщение.

5. *Если селектор не найден*, нужно искать в новом словаре сообщений (возврат на шаг 3). Новый словарь сообщений находится в надклассе последнего класса в словаре котором происходил поиск метода. Этот цикл может повториться несколько раз, идя вверх по цепи надклассов.

Если селектор не найден ни в классе получателя и ни в одном из его надклассов, сообщается об ошибке, выполнение байткодов следующих за посылкой приостанавливается.

Посылка надклассу

Вариант байткода посылки называемый посылки наду использует для нахождения *Откомпилированного метода*, связанного с сообщением, немного отличающийся алгоритм. Все действия те же за исключением второго шага, который находит начальный словарь для поиска сообщения. Когда встречается посылка наду, используется следующий второй шаг:

2. *Находится словарь сообщений.* Начальный словарь сообщений находится в надклассе класса в котором находится текущий выполняемый *Откомпилированный метод*.

Байткоды посылки наду используются когда в качестве получателя сообщения в исходном методе используется *над*. Байткод используемый для помещения получателя будет тем же самым как при использовании переменной *сам*, но будет использоваться байткод посылки наду для описания селектора.

В качестве использования посылки наду рассмотрим воображаемый подкласс *Прямоугольника* называемый *Прямоугольник с тенью* который добавляет переменную экземпляра *тень*. *Прямоугольник* может отвечать на сообщения *тень:* создавая новый *Прямоугольник с тенью*. *Прямоугольник с тенью* предоставляет новый метод для сообщения *пересечь:*, возвращающий *Прямоугольник с тенью* вместо *Прямоугольника*. Этот метод должен использовать *над* для доступа к своей возможности вычислять пересечение.

```
пересечь: прямоугольник
  ↑ ( над пересечь: прямоугольник ) тень: тень.
```

Прямоугольник с тенью **пересечь:**

112	поместить на стэк получателя (себя)
16	поместить на стэк первую временную переменную (прямоугольник)
133, 33	послать *наду* одноаргументное сообщение с селектором из второй позиции блока литералов (пересечь:)
2	поместить на стэк значение третьей переменной экземпляра получателя (тень)

224	послать одноаргументное сообщение с первым селектором из блока литералов (тень:)
124	возвратить объект с вершины стэка как значение сообщения (пересечь:)

блок литералов
#тень:
#пересечь:
Ассоциация: #Прямоугольник с тенью → Прямоугольник с тенью

Важно отметить что начальный класс поиска в ответ на посылку *наду* будет надклассом класса получателя только если *Откомпилированный метод* содержащий посылку *наду* находится в классе получателя. Если *Откомпилированный метод* находится в надклассе класса получателя, то поиска будет начат в надклассе этого класса. Т.к. состояние интерпретатора не содержит класса в котором находится каждый *Откомпилированный метод*, то эта информация включается в сам *Откомпилированный метод*. Каждый *Откомпилированный метод*, содержащий байткод посылки *наду*, ссылается на класс в чьём словаре он находится. Последнее положение в блоке литералов такого *Откомпилированного метода* содержит ассоциацию указывающую на класс.

26.2.4 Элементарные методы

Действия интерпретатора, после нахождения *Откомпилированного метода* зависят от того указано ли в *Откомпилированном методе* что на сообщение может ответить элементарный метод. Если элементарный метод не указан, то создаётся новый *Контекст метода* и он становится активным, как описано в предыдущих разделах. Если в *Откомпилированном методе* указан элементарный метод, то интерпретатор может ответить на сообщение без выполнения байткодов. Например, один из элементарных методов связан с сообщением + экземпляров *Малого целого*.

+ слагаемое
<элементарный: 1>

↑над + слагаемое.

Малое целое + связан с элементарным методом 1

112	поместить на стэк получателя (себя)
16	поместить на стэк первую временную переменную (аргумент слагаемое)
133, 32	послать *наду* одноаргументное сообщение с селектором из первой позиции блока литералов ()
124	возвратить объект с вершины стэка как значение сообщения (+)

блок литералов
#+

Даже если для *Откомпилированного метода* указан элементарный метод интерпретатор, возможно, не сможет ответить успешно. Например аргументом сообщения + может быть не другой экземпляр *Малого целого* или сумма может быть не представима *Малым целым*. Если интерпретатор не может, по некоторой причине, выполнить элементарный метод говорят что элементарный метод провалился. Когда элементарный метод проваливается выполняются байткоды *Откомпилированного метода* так как будто элементарный метод не был указан. Метод + *Малого целого* указывает что при провале элементарного метода нужно использовать метод + надкласса (*Целое*).

В системе есть примерно сотня элементарных методов которые могут выполнять четыре типа операций. Точная функция всех элементарных методов будет описана в Главе 29.

1. Арифметика.
2. Управление хранилищем объектов.
3. Управление.
4. Ввод-вывод.

26.3 Память объектов

Память объектов предоставляет интерпретатору интерфейс к к объектам составляющим виртуальный образ Смолтока. Каждый объект связан с уникальным идентификатором называемым указателем объекта. Память объектов и интерпретатор обмениваются информацией об объектах при помощи указателей объектов. Размер указателей объектов задаёт максимальное количество объектов доступных системе Смолток. Это число никак не фиксировано в языке, но реализация описанная в этой книге использует 16-ти битные указатели объектов, допускающие ссылки на 65536 объектов. Реализация системы Смолток с большими ссылками на объекты потребует изменения некоторых частей определения виртуальной машины. Детали связанные с этим вопросом не являются темой этой книги.

Память объектов связывает каждый указатель объекта с набором других указателей объектов. Каждый указатель объекта связан с указателем объекта класса. Если у объекта есть переменные экземпляра, то их указатели объектов также связаны с указателями объектов их значений. Ссылкой на конкретную переменную экземпляра является номер относительно нуля. Значение переменной экземпляра может быть изменено, но класс связанный с объектом изменить нельзя. Память объектов предоставляет интерпретатору следующие пять основных функций:

1. *Доступ* к значению *переменной экземпляра* объекта. Требуется задать указатель объекта и номер переменной экземпляра. Возвращается указатель объекта значения переменной экземпляра.

2. *Изменение* значения *переменной экземпляра* объекта. Требуется задать указатель объекта и номер переменной экземпляра. Также нужно задать указатель объекта нового значения.

3. *Доступ* к *классу* объекта. Требуется задать указатель объекта экземпляра. Возвращается указатель объекта на класс экземпляра.

4. *Создание* нового *объекта*. Требуется задать указатель объекта на класс нового объекта и количество переменных экзем-

пляра. Возвращается укзатель объекта на новый экземпляр.

5. Определение *количества переменных экземпляра* объекта. Нужно задать указатель объекта. Возвращается количество переменных экземпляра.

Нет явной функции памяти объектов удаляющей не нужные объекты т.к. эти объекты очищаются автоматически. Объект очищается когда на него из других объектов больше не существует указателей. Эта очистка может выполняться при помощи подсчёта ссылок или сборки мусора.

Есть два дополнительных свойства памяти объектов которые предоставляют эффективное представление числовой информации. Первое свойство это набор специальных указателей объектов для экземпляров класса *Малое целое*. Второе позволяет объектам содержать целые значения вместо указателей объектов.

Представление малых целых

Экземпляры класса *Малое целое* представляют целые от −16384 до 16383. Каждому из этих экземпляров соответствует уникальный указатель объекта. Все эти указатели объектов содержат 1 в младшем двоичном разряде и в 15 старших битах содержат представление своего значения в виде дополнения до двух. Экземпляры *Малого целого* не нуждаются в памяти экземпляра т.к. и класс и значение можно определить из указателя объекта. Памятью объектов предоставляется две дополнительные функции для конвертации между указателями объектов *Малого целого* и числовыми значениями.

6. Поиск численного значения представляемого *Малым целым*. Требуется задать указатель объекта *Малого целого*. Возвращается значение в виде дополнения до двух.

7. Поиск *Малого целого* представляющего численное значение. Требуется задать значение в виде дополнения до двух. Возвращается указатель объекта на *Малое целое*.

Это представление *Малых целых* предполагает что в системе может существовать 32768 экземпляров других классов. Оно также

предполагает что равенство (=) и эквивалентность (==) должны быть одним и тем же для экземпляров *Малого целого*. Экземпляры вне диапазона от −16384 до 16383 представляются экземплярами классов *Большое положительное целое* и *Большое отрицательное целое*. У них может быть несколько экземпляров представляющих то же значение, но равенство и эквивалентность будут различными.

Наборы целых значений

Есть другое специальное представление для объектов представляющих наборы целых. Вместо хранения указателей объектов представляющих *Малые целые* содержащихся в наборе, хранятся действительные численные значения. Значения в этих специальных наборах рассматриваются как положительные. Есть два варианта наборов, одни ограничивают значения числами меньшими чем 256 а другие числами меньшими чем 65536. Память объектов предоставляет функции аналогичные первым пяти указанным выше, но для объектов чьё содержимое это численные значения а не указатели объектов.

Различие между объектами содержащими указатели объектов и объектами содержащими целые значения никогда не видно программисту. Когда при помощи посылки сообщения осуществляется доступ к одному из этих специальных наборов целых, то возвращается указатель объекта представляющий значение. Основное назначение этих специальных наборов в том что они могут отвергать попытки сохранения целых объектов не содержащихся в правильных пределах.

26.4 Оборудование

Реализация Смолтока была описана как виртуальная машина чтобы избежать ненужных зависимостей от оборудования. Естественно предполагать что оборудование состоит из исполнителя и достаточного объёма памяти для хранения виртуального образа и машинных процедур моделирующих интерпретатор и память объектов. Текущий размер виртуального образа требует хотя бы пол мегабайта

памяти.

Размер исполнителя и организация памяти не задаётся определением виртуальной машины. Т.к. указатели объектов 16 битные, то наиболее подходящим будет 16 битный исполнитель и память с 16 битными словами. Как с исполнителями и памятью любой системы, чем быстрее тем лучше.

Другие требования к оборудованию задают элементарные методы от которых зависит виртуальный образ. Это устройства ввода-вывода и часы которые указаны ниже.

1. Растровый дисплей. Наиболее подходящим случаем будет если растр может располагаться в памяти объектов, но это не абсолютно необходимо.

2. Указывающее устройство.

3. Три кнопки, связанные с указывающим устройством. Лучше всего чтобы они располагались на самом указывающем устройстве.

4. Клавиатура, либо с кодировкой АСКОИ либо undecoded ALTO.

5. Диск. Стандартный виртуальный образ Смолтока содержит только скелет системы диска которая должна быть подогнана к фактически используемому диску.

6. Миллисекундный таймер.

7. Часы реального времени с разрешением в секунду.

Глава 27

Определение виртуальной машины

Оглавление

Глава 26 описывает функции виртуальной машины Смолтока, которая состоит из интерпретатора и памяти объектов. Эта и три следующие главы представляют более формальное определение этих двух частей виртуальной машины. Большинство реализаций виртуальной машины делаются на машинном языке или микрокоде. Однако, для целей определения, эти главы будут представлять реализацию виртуальной машины на самом Смолтоке. Т.к. это круговое определение, все усилия были направлены на то чтобы в результате не осталось скрытых деталей.

Эта глава состоит из трёх разделов. Первый описывает соглашения и терминологию используемую в формальном определении. Он также содержит несколько предупреждений о возможных недоразумениях могущих возникнуть из за формы этого определения.

Второй раздел описывает функции памяти объектов используемые интерпретатором. Реализация этих функций будет описана в Главе 30. Третий раздел описывает три главных объекта которыми манипулирует интерпретатор: методы, контексты и классы. Глава 28 описывает набор байткодов и то как они интерпретируются, Глава 29 описывает элементарные функции.

27.1 Форма определения

Два класс описания с именами *Интерпретатор* и *Память объектов* составляют формальное определение виртуальной машины Смолтока. Реализация *Интерпретатора* будет в деталях описана в этой и следующих двух главах; реализация *Памяти объектов* в Главе 30.

Потенциальный источник путаницы в этой главе возникает из за двух систем Смолток вовлечённых в рассмотрение, система содержащая *Интерпретатор* и *Память объектов* и система которая интерпретируется. *Интерпретатор* и *Память объектов* содержат методы и переменные экземпляра и они также манипулируют методами и переменными экземпляра их интерпретирующей системы. Для минимизации путаницы будет разделяться терминология для каждой системы. Методы *Интерпретатора* и *Памяти объектов* будут называться процедурами; термин метод будет зарезервирован для интерпретируемых методов. Также, переменные экземпляра *Интерпретатора* и *Памяти объектов* будут называться регистрами; термин переменная экземпляра будет зарезервирован для переменных экземпляра объектов интерпретируемой системы.

Аргументами процедур и содержимым регистров *Интерпретатора* и *Памяти объектов* всегда будут экземпляры *Целого* (*Малые целые* и *Большие положительные целые*). Это тоже может быть источником путаницы т.к. есть *Целые* в интерпретируемой системе. *Целые* являющиеся аргументами для процедур и содержимым регистров представляют указатели объектов и численные значения интерпретируемой системы. Некоторые из них будут представлять указатели объектов или значения *Целых* в интерпретируемой системе.

Процедуры интерпретатора в этом определении будут представлены в форме определения методов Смолтока. Например:

```
имя процедуры: имя аргумента
  | временная переменная |
  временная переменная ← сам другая процедура: имя аргумента.
  ↑ временная переменная − 1.
```

Процедуры определения будут содержать пять типов выражений.

1. *Вызовы других процедур интерпретатора.* Т.к. оба вызова и определение процедур находятся в *Интерпретаторе*, то они будут сообщениями *себе*.

   - сам заголовок для: новый метод
   - сам
     сохранить значение указателя инструкции: значение
     в контекст: указатель контекста

2. *Вызовы процедур памяти объектов.* *Интерпретатор* использует имя *память* для ссылок на свою память объектов, поэтому эти вызовы будут сообщениями *памяти*.

   - память извлечь класс: новый метод
   - память
     сохранить указатель: номер отправителя
     в объект: указатель контекста
     со значением: активный контекст

3. *Арифметические операции над указателями объектов и численными значениями.* Арифметические операции будут представляться обычными арифметическими выражениями Смолтока, поэтому они будут сообщениями самим числам.

   - значение получателя + значение аргумента
   - указатель селектора сдвинуть биты: −1

4. *Доступ к рядам.* Таблицы которыми управляет интерпретатор представлены в формальном определении как *Ряды*. Доступ к ним будет представляться в виде сообщений *от:* и *от:пом:*.

   - кэш методов от: хэш
   - список семафоров от: номер семафора пом: указатель семафора

5. *Условные управляющие структуры.* Управляющие структуры виртуальной машины будут представляться стандартными управляющими структурами Смолтока. Условный выбор будет представляться сообщениями *Логике*. Условные повторения будут представляться сообщениями блокам.

   - номер < длина истина: [. . .]
   - флаг размера = 1 истина: [. . .] ложь: [. . .]
   - [текущий класс ~= указатель пусто.] пока истина: [. . .]

Определение *Интерпретатора* описывает функцию интерпретатора байткодов Смолтока; однако, форма машинного языка реализации интерпретатора может быть совсем другой, особенно в используемых управляющих структурах. Выбор подходящей процедуры для исполнения байткода это пример того что машинный язык может делать по другому. Для нахождения выполняемой процедуры на машинном языке, скорей всего, нужно было бы вычислить прыжок при помощи некоторых арифметических вычислений; тогда как, как мы увидим, *Интерпретатор* делает серию проверок и вызовы процедур. В реализации на машинном языке процедура выполняющая байткоды, при завершении просто бы делала прыжок назад в начало выбора байткода, вместо возвращения при помощи выхода из процедуры.

Ещё одно различие между *Интерпретатором* и реализацией на машинном языке заключается в степени оптимизации кода. Ради ясности, процедуры определённые в этой главе не оптимизированы. Например, чтобы выполнить задачу, *Интерпретатор* может прочитать указатель из памяти объектов несколько раз в различных процедурах, в то время как более оптимизированный интерпретатор

мог бы для дальнейшего использования сохранить значение в регистре. Многие процедуры в формальном определении могли бы не быть процедурами в реализации на машинном языке, вместо этого они бы были записаны непосредственно в текст другой процедуры.

27.2 Интерфейс памяти объектов

Глава 26 дала неформальное описание памяти объектов. Т.к. процедурам *Интерпретатора* нужно взаимодействовать с памятью объектов, то требуется формальное функциональное определение. Оно будет дано как определение протокола класса *Память объектов*. Глава 30 опишит один из способов реализации этого определения протокола.

Память объектов связывает 16-ти битные указатели объектов с:

1. указателем объекта на класс описывающий объект и
2. набором 8- или 16-битных полей содержащих указатели объектов или численные значения.

Интерфейс памяти объектов использует для указания полей объекта целые номера относительно нуля. В интерфейсе между интерпретатором и памятью объектов используются экземпляры *Целого* и для указателей объектов и для номеров полей.

Протокол *Памяти объектов* содержит пары сообщений для чтения и сохранения по указателям объектов или численным значениям полей объекта.

Протокол экземпляров *Памяти объектов*

доступ через указатель объекта

достать указатель: номер поля из объекта: указатель объекта

Возвращает указатель объекта находящийся в поле номер поля в объекте связанном с указателем объекта.

сохранить указатель: номер поля в объект: указатель объекта со значением: значение указателя

Сохраняет значение указателя объекта в поле номер поля объекта связанного с указателем объекта.

доступ к словам

достать слово: **номер поля** **из объекта:** **указатель объекта**
Возвращает 16-ти битное численное значение находящиеся в поле номер поля в объекте связанном с указателем объекта.

сохранить слово: **номер поля** **в объект:** **указатель объекта** **со значением:** **слово значение**
Сохраняет 16-ти битное численное значение слово значение в поле номер поля в объекте связанном с указателем объекта.

доступ к байтам

достать байт: **номер байта** **из объекта:** **указатель объекта**
Возвращает 8-ми битное численное значение находящиеся в байте номер байта в объекте связанном с указателем объекта.

сохранить байт: **номер байта** **в объект:** **указатель объекта** **со значением:** **байт значение**
Сохраняет 8-ми битное численное значение байт значение в байт номер байта в объекте связанном с указателем объекта.

Заметьте что *достать указатель:из объекта:* и *достать слово:из объекта:* будут, возможно, реализованы идентичным образом, т.к. они оба извлекают 16-ти битные значения. Однако, реализации *сохранить указатель:из объекта:со значением:* и *сохранить слово:из объекта:со значением:* будут различаться т.к. они должны осуществлять подсчёт ссылок (см. Главу 30) если память объектов поддерживает динамический подсчёт ссылок. Отдельный интерфейс для *достать указатель:из объекта:* и *достать слово:из объекта:* поддерживается для симметричности.

Несмотря на то что поддержание подсчёта ссылок может быть выполнено автоматически в процедуре *сохранить указатель:из объекта:со значением:*, есть несколько случаев в которых процедуры интерпретатор должны напрямую манипулировать количеством ссылок. Поэтому в интерфейс памяти объектов включены следующие две процедуры. Если память объектов использует для очистки недоступных объектов только сборку мусора, то эти процедуры ничего не делают.

Протокол экземпляров *Памяти объектов*

подсчёт ссылок

увеличить ссылки на: указатель объекта
Добавляет единицу к количеству ссылок на объект с указателем объекта.

уменьшить ссылки на: указатель объекта
Вычитает единицу из количества ссылок на объект с указателем объекта.

Т.к. каждый объект содержит указатель объекта на описание своего класса, то этот указатель считается содержимым одного из полей объекта. Однако в отличии от других полей класс объекта можно извлекать, но его значение невозможно изменить. Из за специального назначения этого указателя к нему осуществляется не так как к другим полям. Поэтому есть специальный протокол для извлечения класса объекта.

Протокол экземпляров *Памяти объектов*

доступ к указателю класса

извлечь класс: указатель объекта
Возвращает указатель объекта на класс описывающий объект связанный с указателем объекта.

Длина объекта также может рассматриваться как содержимое одного из его полей. Однако подобно полю класса это поле не может быть изменено. В памяти объектов есть два сообщения спрашивающие количество слов объекта и количество байтов объекта. Заметьте что не делается различия между словами и указателями т.к. предполагается что они оба занимают точно одно поле.

Протокол экземпляров *Памяти объектов*

доступ к длине

извлечь длину в словах: указатель объекта
Возвращает количество полей в объекте связанном с указателем объекта.

извлечь длину в байтах: указатель объекта

Возвращает количество полей байтов в объекте связанном с указателем объекта.

Другая важная работу памяти объектов — создавать новые объекты. Памяти объектов нужно указать класс и длину и она вернёт указатль на новый объект. Опять есть три версии создаваемых объектов с указателями, со словами или с байтами.

Протокол экземпляров *Памяти объектов*

создание объектов

экземпляр класса: указатель класса с указателями: размер экземпляра

Создаёт новый экземпляр класса чей указатель объекта это указатель класса с количество полей размер экземпляра содержащих указатели. Возвращает указать на вновь созданный объект.

экземпляр класса: указатель класса со словами: размер экземпляра

Создаёт новый экземпляр класса чей указатель объекта это указатель класса с количество полей размер экземпляра содержащих 16-ти битные значения. Возвращает указать на вновь созданный объект.

экземпляр класса: указатель класса с байтами: размер экземпляра в байтах

Создаёт новый экземпляр класса чей указатель объекта это указатель класса с пространством для хранения количество полей в байтах 8-ми битных значений. Возвращает указать на вновь созданный объект.

Две процедуры памяти объектов позволяют перечислять экземпляры класса. Это перечисление происходит в произвольном порядке указателей объектов. Разумно использовать порядок самих указателей.

Протокол экземпляров *Памяти объектов*

перебор экземпляров

первый экземпляр: **указатель класса**
Возвращает указатель объекта первого, в определённом порядке (т.е. указатель объекта с наименьшим значением), экземпляра класса с указателем указатель класса .

экземпляр после: **указатель объекта**
Возвращает указатель объекта следующего, в определённом порядке (т.е. следующий указатель объекта с большим значением), экземпляра того же класса что и объект на который указывает указатель объекта

Следующая процедура памяти объектов позволяет обменять значения двух указателей объектов.

Протокол экземпляров *Памяти объектов*

обмен указателей

обменять указатель: **первый указатель** **и:** **второй указатель**
Делает первый указатель ссылающимся на объект чьим указателем был второй указатель и делает второй указатель ссылающимся на объект чьим указателем был первый указатель.

Как было описано в Главе 26, целые между −16384 и 16383 представляются непосредственно указателями объектов с 1 в младшем разряде и соответствующим значением в форме дополнения до двух в 15-ти старших битах. Такие объекты являются экземплярами *Малого целого*. Значение *Малого целого*, которое обычно должно храниться в поле, в действительности находится из указателя объекта. Поэтому вместо сохранения значения в поле *Малого целого* интерпретатор должен запросить указатель на *Малое целое* с соответствующим значением (используя процедуру *объект целое для:*). И вместо того чтобы извлекать значения из поля, он должен запросить значение связанное с указателем объекта (используя процедуру *целое значение для:*). Также есть две процедуры определяющие ссылается ли указатель объекта на *Малое целое* (*это объект целое:*) и входит ли значение в границы представления *Малого целого* (*это значение целого:*). Функция процедуры *это объект целое:* также

может быть выполнена путём запроса класса объекта и проверки является ли он *Малым целым*.

Протокол экземпляров *Памяти объектов*

доступ к целым

целое значения для: **указателя объекта**
Возвращает значение экземпляра *Малого целого* чей указатель это указатель объекта.

объект целое для: **значения**
Возвращает указатель объекта на экземпляр *Малого целого* со значением значение.

это объект целое: **указатель объекта**
Отвечает *истина* если указатель объекта является экземпляром *Малого целого*, иначе отвечает *ложь*.

это значение целого: **значение**
Отвечает *истина* если значение может быть представлено экземпляром *Малого целого*, иначе отвечает *ложь*.

Интерпретатор предоставляет две специальных процедуры для доступа к полям содержащим *Малые целые*. Процедура *достать целое: из объекта:* возвращает значение *Малого целого* чей указатель хранится в указанном поле. Чтобы удостовериться что указатель ссылается на *Малое целое* производится проверка позволяющая запрашиваемую полю быть не *Малым целым*. Процедура *неудача элементарного метода* будет описана в разделе об элементарных процедурах.

```
достать целое: номер поля из объекта: указатель объекта
  | указатель целого |
  указатель целого ← память достать указатель: номер поля из объекта: указатель объекта.
  ( память это объект целое: указатель целого )
    истина: [↑ память целое значение для: указатель целого. ]
    ложь: [↑ сам неудача элементарного метода. ].
```

Процедура *поместить целое:в объект:со значением:* сохраняет указатель на *Малое целое* с заданным значением в заданное поле.

```
поместить целое: номер поля
в объект: указатель объекта
со значением: значение целого
  | указатель целого |
  ( память это значение целого: значение целого )
    истина: [
     указатель целого ← память объект целое для: значение целого.
      память
        поместить указатель: номер поля
        в объект: указатель объекта
        со значением: указатель целого. ]
    ложь: [ сам неудача элементарного метода. ].
```

Также интерпретатор предоставляет процедуру для переноса нескольких указателей из одного объекта в другой. Она получает в качестве аргументов количество перемещаемых указателей, номер первого поля и указатель исходного объекта и объекта назначения.

```
перенести: количество
от номера: первый из
из объекта: из УО
в номер: первый в
в объект: в УО
  | номер из номер в последний из уо |
  номер из ← первый из.
  последний из ← первый из + количество.
  номер в ← первый в.
  [ номер из < последний из. ]
    пока истина: [
      уо ← память достать указатель: номер из из объекта: из УО.
      память сохранить указатель: номер в в вобъект: в УО со значением: уо.
      память
        сохранить указатель: номер из
        в вобъект: из УО
        со значением: Пустой указатель.
      номер из ← номер из + 1.
      номер в ← номер в + 1. ].
```

Также интерпретатор предоставляет процедуры для извлечения битовых полей из численных значений. Эти процедуры ссылаются на старший бит при помощи номера 0 и на младший бит при помощи номера 15.

```
извлечь биты от: номер первого бита до: номер последнего бита из: целое
  ↑ ( целое сдвинуть биты: номер последнего бита − 15 )
    побитовое и: ( 2 в степени: номер последнего бита − номер первого бита + 1 ) − 1.

старший байт: целое
  ↑ сам извлечь биты от: 0 до: 7 из: целое.

младший байт: целое
  ↑ сам извлечь биты от: 8 до: 15 из: целое.
```

27.3 Объекты используемые интерпретатором

Этот раздел описывает то что может быть названо структурами данных интерпретатора. Несмотря на то что это объекты, и следовательно являются больше чем структурами данных, интерпретатор рассматривает эти объекты как структуры данных. Две первых вида объектов соответствуют структурам данных присутствующих в интерпретаторах большинства языков. *Методы* соответствуют программам, подпрограммами и процедурам. *Контекст* соответствует фрейму стэка или записи активации. Последняя структура описанная в этом разделе, *класс*, используется в основном компилятором. Классы соответствуют объявлениям типов некоторых других языков. Из за сущности сообщений Смолтока, классы должны использоваться интерпретатором во время выполнения программы.

В формальное определение включено много констант. Большинство из них представляют указатели известных объектов и номера полей для некоторых видов объектов. Большинство констант поименовано и процедура инициализирующая их будет включена в качестве определения их значений. Например, следующая процедура

инициализирует указатели объектов известных интерпретатору.

инициализировать малые целые
"Малые целые"
Указатель на минус еденицу ← 65535.
Указатель на ноль ← 1.
Указатель на еденицу ← 3.
Указатель на двойку ← 5.

инициализировать гарантированные указатели
Указатель на пусто ← 2.
Указатель на ложь ← 4.
Указатель на истину ← 6.
Указатель на Ассоциацию планировщика ← 8.
Указатель на класс Цепь ← 14.
Указатель на класс Ряд ← 16.
Указатель на класс Контекст метода ← 22.
Указатель на класс Контекст блока ← 24.
Указатель на класс Точка ← 26.
Указатель на класс Большое положительное целое ← 28.
Указатель на класс Сообщение ← 32.
Указатель на класс Знак ← 40.
Селектор не понимаю ← 42.
Селектор невозможно вернуть ← 44.
Селектор должен быть логическим ← 52.
Указатель на специальные селекторы ← 48.
Указатель на таблицу знаков ← 50.

27.3.1 Откомпилированные методы

Байткоды, выполняемые интерпретатором, находятся в экземплярах *Откомпилированного метода*. Байткоды хранятся как 8-ми битные значения, два в слове. В дополнение к байткодам *Откомпилированный метод* содержит несколько указателей объектов. Первый из этих указателей объектов называется заголовком метода а оставшиеся из указателей объектов составляют блок литералов. Рисунок 27.1 показывает структуру *Откомпилированного*

метода и следующая процедура инициализирует номер используемые для доступа к полям *Откомпилированного метода*.

Рис. 27.1

инициализировать номера метода
"Класс Откомпилированный метод"
Номер заголовка ← 0.
Начало литералов ← 1.

Заголовок это *Малое целое* которое кодирует информацию об *Откомпилированном методе*.

заголовок: указатель метода
↑ память достать указатель: Номер заголовка из объекта: указатель метода.

Блок литералов содержит указатели объектов на которые ссылаются байткоды. Он содержит селекторы сообщений посылаемые методом, разделяемые переменные и константы на которые ссылается метод.

литерал: смещение из метода: указатель метода
↑ память
извлечь указатель: смещение + Начало литералов
из обхекта: указатель метода.

За заголовком и литералами идут байткоды метода. Методы это

единственный вид объектов в системе Смолток который хранить и указатели объектов (в заголовке и блоке литералов) и численные значения (в байткодах). Форма байткодов будет обсуждаться в следующей главе.

Заголовок метода

Т.к. заголовок метода это *Малое целое*, то его значение кодируется в его указателе. 15 старших битов указателя доступны для кодирования информации; младший бит должен быть единицей для указания того что это указатель на *Малое целое*. В заголовок входят четыре поля битов указывающих информацию об *Откомпилированном методе*. Рисунок 27.2 показывает поля битов заголовка.

Рис. 27.2

Количество временных указывает число временных переменных используемых *Откомпилированным методом*. В это количество включено число аргументов.

количество временных: **указатель метода**
↑ сам извлечь биты от: 3 до: 7 из: (сам заголовок: указатель метода).

Флаг большого контекста указывает нужный размер *Контекста метода* из двух возможных. Флаг указывает что сумма максимальной глубины стэка и количества временных переменных больше двенадцати. У меньшего *Контекста метода* пространство под стэк равно 12-ти, а у большего оно равно 32-м.

флаг большого контекста: **указатель метода**
↑ сам извлечь биты от: 8 до: 8 из: (сам заголовок: указатель метода).

Количество временных указывает размер блока литералов *Контекста метода*. Поэтому это поле указывает где начинаются байткоды *Контекста метода*.

количество литералов: **указатель метода**
↑сам количество литералов заголовка: (сам заголовок: указатель метода).

количество литералов заголовка: **указатель заголовка**
↑сам извлечь биты от: 9 до: 14 из: указатель заголовка.

Количество указателей объектов указывает общее число указателей объектов в *Контексте метода*, включая заголовок и блок литералов.

количество указателей объектов: **указатель метода**
↑(сам количество литералов: указатель метода) + Начало литералов.

Следующая процедура возвращает номер байта первого байткода *Откомпилированного метода*.

начальный указатель исктрукции метода: **указатель метода**
↑(сам количество литералов: указатель метода) + Начало литералов * 2 + 1.

Значение флага используется для кодирования количества аргументов которые требуются *Откомпилированному методу* и связан ли с ним элементарный метод.

значение флага: **указатель метода**
↑сам извлечь биты от: 0 до: 2 из: (сам заголовок: указатель метода).

Восемь возможных значений флага имеют следующее значение:

Значение флага	Смысл
0-4	Нет элементарного метода и от 0 до 4 аргументов.
5	Элементарный метод возвращающий себя (0 аргументов).

6	Элементарный метод возвращающий переменную экземпляра (0 аргументов).
7	Расширение заголовка содержит количество аргументов и номер элементарного метода.

Т.к. большинство *Откомпилированных методов* содержит четыре или меньше аргументов и не связаны с элементарным методом, то обычно значение флага это количество аргументов.

Специальные элементарные методы

Методы Смолтока которые только возвращают получателя сообщения () порождают *Откомпилированные методы* без литералов и байткодов, есть только заголовок с флагом равным пяти. Подобным образом методы Смолтока которые только возвращают значение одной из переменных экземпляра получателя порождают *Откомпилированные методы* только из заголовка с флагом равным шести. Все другие методы порождают *Откомпилированные методы* с байткодами. Когда флаг равен шести, то номер возвращаемой переменной экземпляра находится в заголовке в поле обычно используемом для указания количества временных переменных используемых *Откомпилированным методом*. Рисунок 27.3 показывает *Откомпилированный метод* который возвращает переменную экземпляра.

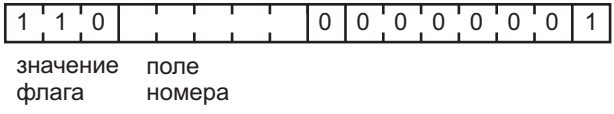


Рис. 27.3

Следующая процедура возвращает номер из поля представляющего возвращаемую переменную экземпляра когда флаг равен шести.

поле номер: указатель метода
↑сам извлечь биты от: 3 до: 7 из: (сам заголовок: указатель метода).

Расширение заголовка метода

Если значение флага равно семи, то предпоследний литерал это расширение заголовка, являющиеся другим *Малым целым*. Расширение заголовка состоит из двух полей битов которые кодируют количество аргументов и номер элементарного метода. Рисунок 27.4 показывает поля битов расширения заголовка.

Рис. 27.4

Следующие процедуры используются для доступа к расширению заголовка и его полям битов.

расширение заголовка: указатель метода
| количество литералов |
количество литералов ← сам количество литералов: указатель метода.
↑сам литерал: количество литералов – 2 из метода: указатель метода.

количество аргументов: указатель метода
| значение флага |
значение флага ← сам значение флага: указатель метода.
значение флага < 5 истина: [↑значение флага.].
значение флага < 7
  истина: [↑0.]
  ложь: [
    ↑сам

```
        извлечь биты от: 2
        до: 6
        из: (сам расширение заголовка: указатель метода).].

номер элементарного метода: указатель метода
  | значение флага |
  значение флага ← сам значение флага: указатель метода.
  значение флага = 7
    истина: [
      ↑сам
        извлечь биты от: 7
        до: 14
        из: (сам расширение заголовка: указатель метода).]
    ложь: [↑0.].
```

Любой *Откомпилированный метод* посылающий сообщение надклассу (т.е. сообщение *наду*) или содержащий расширение заголовка, содержит в качестве последнего литерала *Ассоциацию* со значением равным классу в котором содержится словарь методов в котором находится этот *Откомпилированный метод*. Это называется классом метода и он вычисляется следующей процедурой.

```
класс метода: указатель метода
  | количество литералов ассоциация |
  количество литералов ← сам количество литералов: указатель метода.
  ассоциация ← сам литерал: количество литералов − 1 из метода: указатель метода.
  ↑память извлечь указатель: Номер значения из объекта: ассоциация.
```

Пример *Откомпилированного метода* с классом в блоке литералов был дан в последней главе. *Откомпилированный метод* сообщения *пересечь:* из класса *Прямоугольник с тенью* был показан в последней главе в разделе Сообщения.

27.3.2 Контексты

Интерпретатор использует контексты для представления состояния выполняемых *Откомпилированных методов* и блоков. Контекст может быть *Контекстом метода* или *Контекстом блока*. *Контекст метода* представляет выполнение *Откомпилированного метода* которое вызвано сообщением. Рисунок 27.5 показывает *Контекст метода* и его *Откомпилированный метод*.

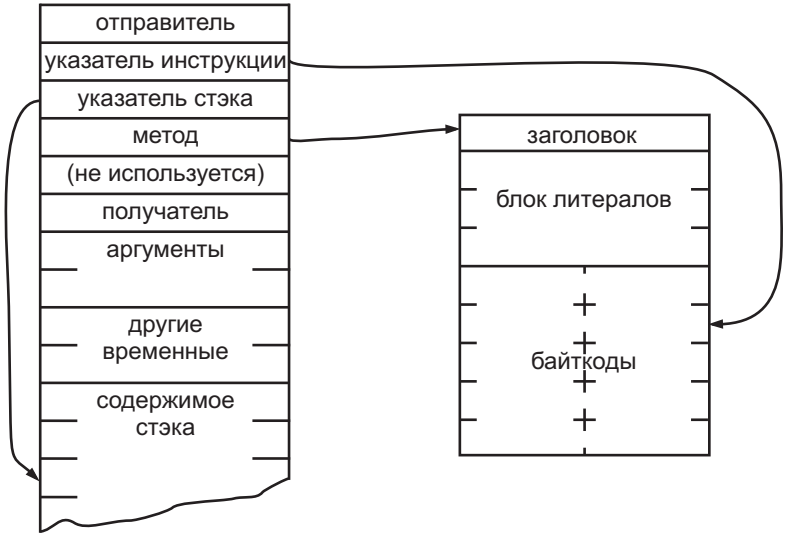


Рис. 27.5

Контекст блока представляет блок встретившийся в *Откомпилированном методе*. *Контекст блока* ссылается на *Контекст метода* чей *Откомпилированный метод* содержит представляемый им блок. Этот контекст называется домом *Контекста блока*. Рисунок 27.6 показывает *Контекст блока* и его дом.

Номера используемые для доступа к полям контекста инициализируются следующей процедурой.

инициализировать номера контекста
Номер отправителя ← 0.

```
Номер указателя инструкции ← 1.
Номер указателя стэка ← 2.
Номер метода ← 3.
Номер получателя ← 5.
Начало блока временных ← 6.
Номер вызвавшего ← 0.
Номер количества аргументов блока ← 3.
Номер начальный ← 4.
Номер дома ← 5.
```

Оба вида контекстов имеют шесть фиксированных поля соответствующих шести именованным переменных экземпляра. За этими фиксированными полями следуют несколько нумерованных поля. Нумерованные поля используются под блок временных (аргументы и временные переменные) за которым следует содержимое стэка. Следующие процедуры используются для извлечения и сохранения указателя инструкции и указателя стэка хранящихся в контексте.

```
указатель инструкции контекста: указатель контекста
    ↑сам достать целое: Номер указателя инструкции из объекта: указатель контекста.

сохранить значение указателя инструкции: значение
в контекст: указатель контекста
  сам
    сохранить целое: Номер указателя инструкции
    в объект: указатель контекста
    со значением: значение.

указатель стэка контекста: указатель контекста
    ↑сам достать целое: Номер указателя стэка из объекта: указатель контекста.

сохранить значение указателя стэка: значение в контекст:
указатель контекста
  сам
    сохранить целое: Номер указателя стэка
    в объект: указатель контекста
    со значением: значение.
```

Контекст блока хранит ожидаемое количество аргументов блока в одном из своих полей.

количество аргументов блока: **указатель блока**
↑сам
достать целое: Номер количества аргументов блока
из объекта: указатель блока.

Котекст представляющий исполняемый в данный момент *Откомпилированный метод* или блок называется *активным контекстом.* Интерпретатор кэширует в своих регистрах наиболее часто используемую часть содержимого активного контекста. Этими регистрами являются:

Регистры интерпретатора связанные с контекстом

активный контекст	Это сам активный контекст. Им может быть либо *Контекст метода* либо *Контекст блока.*
домашний контекст	Если активный контекст является *Контекстом метода*, то домашний контекст это тот же самый контекст. Если активный контекст является *Контекстом блока*, то домашний контекст это содержимое поля дом активного контекста. Этот регистр всегда содержит *Контекст метода.*
метод	Это *Откомпилированный метод* содержащий байткоды выполняемые интерпретатором.
получатель	Это объект получивший сообщение которое выполняется методом домашнего контекста.
указатель инструкции	Это номер байта следующего исполняемого байткода метода.
указатель стэка	Это номер поля активного контекста содержащего вершину стэка.

При изменении активного контекста (когда вызывается новый *Откомпилированный метод*, когда происходит возврат из *Откомпилированного метода* или когда происходит переключения

процесса), все эти регистры должны быть обновлены при помощи следующей процедуры.

извлечь регистры контекста
(сам это контекст блока: активный контекст)
истина: [
домашний контекст ← память извлечь указатель: Номер дома из объекта: активный контекст.]
ложь: [домашний контекст ← активный контекст.].
получатель ← память извлечь указатель: Номер получателя из объекта: домашний контекст.
метод ← память извлечь указатель: Номер метода из объекта: домашний контекст.
указатель инструкции ← (сам указатель инструкции контекста: активный контекст) − 1.
указатель стэка ← (сам указатель стэка контекста: активный контекст) + Начало блока временных − 1.

Заметьте что получатель и метод извлекаются из *домашнего контекста* а *указатель инструкции* и *указатель стэка* из *активного контекста*. Интерпретатор различает *Контекст метода* и *Контекст блока* по тому что в одном и том же поле *Контекст метода* хранит указатель метода (указатель на объект) а *Контекст блока* хранит число аргументов блока (указатель на целое). Если в данном положении хранится указатель на целое, то класс контекста — *Контекст блока*; иначе класс — *Контекст метода*. Распознавание может быть основано на классе контекста, но при таком подходе нужно отдельно рассматривать случаи подклассов *Контекста метода* и *Контекста блока*.

это контекст блока: указатель контекста
| метод или аргументы |
метод или аргументы ← память извлечь указатель: Номер метода из объекта: указатель контекста.
↑ память это объект целое: метод или аргументы.

Перед тем как контекст станет активным, нужно сохранить при помощи следующей процедуры значения указателя инструкций и указателя стэка в активный контекст.

```
сохранить регистры контекста
  сам
    сохранить значение указателя инструкции: указатель инструкции + 1
      в контекст: активный контекст.
  сам
    сохранить значение указателя стэка: указатель стэка — Начало блока временных + 1
      в контекст: активный контекст.
```

Значение других регистров не изменяется поэтому их не нужно сохранять в контекст. Указатель инструкции хранящийся в контексте это номер поля метода относительно единицы т.к. подномера в Смолтоке (т.е. сообщение *от:*) используют номера относительно единицы. Однако память использует номера относительно нуля; поэтому процедура *извлечь регистры контекста* вычитает единицу для преобразования их в номера памяти а процедура *сохранить регистры контекста* обратно добавляет единицу. Указатель стэка хранимый в контексте показывает как далеко находится вершина стэка выполнения под фиксированными полями контекста (т.е. насколько далеко после начала блока временных) потому что подномера в Смолтоке принимают во внимание фиксированные поля и извлекаются из нумерованных полей следующих за фиксированными. Однако память требует номера относительно начала объекта; поэтому процедура *извлечь регистры контекста* добавляет смещение начала блока временных (константу) а процедура *сохранить регистры контекста* вычитает это смещение.

Следующие процедуры производят различные операции на стэке активного контекста.

```
протолкнуть: объект
  указатель стэка ← указатель стэка + 1.
  память
    сохранить указатель: указатель стэка
    в объект: активный контекст
    со значением: объект.

вытолкнуть стэк
```

```
| вершина стэка |
вершина стэка ← память извлечь указатель: указатель стэка из объекта: активный контекст.
указатель стэка ← указатель стэка − 1.
↑ вершина стэка.
```

вершина стэка

```
↑ память извлечь указатель: указатель стэка из объекта: активный контекст.
```

значение стэка: смещение

```
↑ память
    извлечь указатель: указатель стэка − смещение
    из объекта: активный контекст.
```

вытолкнуть: число

```
указатель стэка ← указатель стэка − число.
```

отменить выталкивание: число

```
указатель стэка ← указатель стэка + число.
```

Регистры активного контекста должны подсчитываться как часть ссылки на память объектов которая освобождает недоступные объекты. Если память объектов поддерживает динамический подсчёт ссылок, то процедура изменяющая активный контекст должна выполнять соответствующий подсчёт ссылок.

новый активный контекст: контекст

```
сам сохранить регистры контекста.
память уменьшить ссылки на: активный контекст.
активный контекст ← контекст.
память увеличить ссылки на: активный контекст.
сам извлечь регистры контекста.
```

Следующие процедуры извлекают поля контекстов нужные интерпретатору не очень часто чтобы кэшировать их в регистрах. Отправитель это контекст в который нужно вернуться когда *Откомпилированный метод* возвращает значение (либо из за "↑"либо из за того что метод закончился). Т.к. явное возвращение значения бло-

ком завершает выполнение охватывающего *Откомпилированного метода*, то отправитель извлекается из домашнего контекста.

отправитель

↑память извлечь указатель: Номер отправителя из объекта: домашний контекст.

Вызвавший это контекст в который нужно вернуться когда *Контекст блока* возвращает значение (при достижении конца блока).

вызвавший

↑память извлечь указатель: Номер отправителя из объекта: активный контекст.

Т.к. временные переменные на которые ссылается блок те же самые что и у охватывающего *Откомпилированного метода*, то временные извлекаются из домашнего контекста.

временная: смещение

↑память
извлечь указатель: смещение + Начало блока временных
из объекта: домашний контекст.

Следующая процедура предоставляет удобный доступ к литералам текущего выполняемого *Откомпилированного метода*.

литерал: смещение

↑сам литерал: смещение из метода: метод.

27.3.3 Классы

В ответ на сообщение интерпретатор находит для выполнения соответствующий *Откомпилированный метод* в словаре методов. Словарь методов хранится в классе получателя сообщения или в одном из его надклассов. Структура класса и связанного с ним словаря методов показаны на рисунке 27.7. Чтобы определить требования экземпляра к памяти, в дополнение к словарю методов и надклассам, интерпретатор использует определение экземпляров класса. Другие поля классов используются только методами Смолтока и игнорируются интерпретатором. Следующая процедура инициализирует но-

мера используемые для доступа к полям классов и их словарю методов.

инициализировать номера класса

Номер надкласса ⟵ 0.
Номер словаря сообщений ⟵ 1.
Номер определения экземпляра ⟵ 2.
Номер ряда методов ⟵ 1.
Начало селекторов ⟵ 2.

Для кэширования состояния поиска метода интерпретатор использует несколько регистров.

Регистры интерпретатора связанные с классами

селектор сообщения	Это селектор посланного сообщения. Он всегда является *Символом*.
количество аргументов	Это количество аргументов посланного сообщения. Он показывает где на стэке находится получатель сообщения т.к. он находится под аргументами.
новый метод	Это метод связанный с *селектором сообщения*.
номер элементарного метода	Это номер элементарной процедуры связанной с *новым методом*, если она есть.

Словарь методов это *Тождественный словарь*. *Тождественный словарь* это подкласс *Множества* с дополнительным содержащим значения связанные с содержимым *Множества*. Селекторы сообщений хранятся в нумерованных переменных экземпляра унаследованных от *Множества*. *Откомпилированные методы* хранятся в *Ряде* добавленном *Тождественным словарём*. Номер *Откомпилированного метода* равен номеру соответствующего селектора в нумерованных экземплярах самого объекта словаря. Номер под которым сохраняется селектор и *Откомпилированный метод* вычисляется хэш функцией.

Селекторы это экземпляры *Символа*, поэтому они могут быть проверены на равенство проверкой равенства их указателей. Т.к. указатели объектов для *Символов* задают эквивалентность, то хэш функция может быть функцией указателя объекта. Т.к. указатели

объектов распределены квазислучайно, то сам указатель объекта представляет хэш функцию. Указатель сдвинутый вправо на один бит будет лучшей хэш функцией, т.к. все указатели объектов, не являющихся *Малыми целыми*, чётны.

```
хэш: указатель объекта
  ↑указатель объекта сдвинуть биты: −1.
```

В поиске по селектору сообщения предполагается что метод метод помещён в словарь с использованием этой же хэш функции. Алгоритм хэширования упрощает исходную хэш функцию по модулю количество нумерованных полей в словаре. Это даёт номер в словаре. Чтобы сделать вычисление упрощения по модулю простым, словарь методов содержит количество полей равное степени двойки. Следовательно вычисление модуля может быть выполнено наложением маски на соответствующее количество бит. Если селектор не найден в начальном положении хэша, то проверяются последующие поля до тех пор пока не будет найден селектор или не встретится *пусто*. Если при поиске встретилось *пусто*, то в словаре нету такого селектора. Если при поиске закончился словарь, то поиск продолжается с первого поля.

Следующая процедура ищет в словаре *Откомпилированный метод* связанный с *Символом* в регистре *селектор сообщения*. Если она находит *Символ*, то она помещает указатель на соответствующий *Откомпилированный метод* в регистр *новый метод*, его номер элементарного метода в регистр *номер элементарного метода* и возвращает *истину*. Если в словаре не находится этот *Символ*, то процедура возвращает *ложь*. Т.к. единственным условием выхода из цикла является нахождение *пусто* или соответствующего *Символа*, то процедура должна проверить весь словарь (т.е. все не пустые элементы). Она делает это сохраняя информацию о том произошёл ли переход через границу. Если переход осуществлён два раза, то в словаре нету селектора.

```
искать метод в словаре: словарь
  | длина номер маска переходил через границу следующий селектор
ряд методов |
  длина ← память извлечь длину в словах: словарь.
  маска ← длина − Начало селекторов − 1.
```

```
  номер ← ( маска побитовое и: ( сам хэш: селектр сообщения ) ) + Начало селекторов.
  переходил через границу ← ложь.
  [ истина. ]
    пока истина: [
        следующий селектор ← память извлечь указатель: номер из объекта: словарь.
      следующий селектор = Указатель на пусто истина: [ ↑ истина. ].
      следующий селектор = селектр сообщения
        истина: [
          ряд методов ← память извлечь указатель: Номер ряда методов из объекта: словарь.
          новый метод ← память
            извлечь указатель: номер – Начало селекторов
            из объекта: ряд методов.
          номер элементарного метода ← сам номер элементарного метода: новый метод.
          ↑ истина. ].
      номер ← номер + 1.
      номер = длина
        истина: [
          переходил через границу истина: [ ↑ ложь. ].
          переходил через границу ← истина.
          номер ← Начало селекторов. ]. ].
```

Эта процедура используется в следующей процедуре для нахождения метода класса связанного с селектором. Если селектор не находится в словаре начального класса, то он ищется в следующем классе цепи надклассов. Поиск идёт по цепи надклассов до тех пор пока не находится метод или не заканчивается цепь надклассов.

```
искать метод в классе: класс
  | текущий класс словарь |
  текущий класс ← класс.
  [ текущий класс ~= Указатель на пусто. ]
    пока истина: [
      словарь ← память
        извлечь указатель: Номер словаря сообщений
```

```
            из объекта: текущий класс.
        ( сам искать метод в словаре: словарь ) истина: [ ↑ истина. ].
        текущий класс ← сам надкласс для: текущий класс. ].
    селектор сообщения = Селектор не понимаю
     истина: [ сам ошибка: 'Recursive not understood error encountered'. ].
    сам создать текущее сообщение.
    селектор сообщения ← Селектор не понимаю.
    ↑ сам искать метод в классе: класс.

надкласс для: указатель класса
    ↑ память извлечь указатель: Номер надкласса из объекта: указатель класса.
```

Интерпретатор должен делать что-то необычное когда класс и надклассы объекта которому послано сообщение не содержат *Откомпилированного метода* связанного с селектором сообщения. В соответствии с философией Смолтока, интерпретатор посылает сообщение. *Откомпилированный метод* для этого сообщения гарантированно существует. Интерпретатор упаковывает текущее сообщение в экземпляр класса *Сообщение* и затем ищет *Откомпилированный метод* связанный с селектором *не понимаю:*. Это *Сообщение* становится единственным аргументом сообщения *не понимаю:*. Сообщение *не понимаю:* определено в *Объекте* с *Откомпилированным методом* сообщающем пользователю об ошибке. Чтобы сделать что-то другое этот *Откомпилированный метод* может быть переопределён в классе пользователя. Из за этого при завершении процедуры *искать метод в классе:* в регистре *новый метод* всегда содержится указатель на *Откомпилированный метод*.

```
создать текущее сообщение
    | ряд аргументов сообщение |
    ряд аргументов ← память
        экземпляр класса: Указатель на класс Ряд
        с указателями: количество аргументов.
    сообщение ← память
        экземпляр класса: Указатель на класс Сообщение
        с указателями: сам размер сообщения.
    память
```

```
    сохранить указатель: Номер селектора сообщения
    в объект: сообщение
    со значением: селектор сообщения.
  память
    сохранить указатель: Номер аргументов сообщения
    в объект: сообщение
    со значением: ряд аргументов.
  сам
    перенести: количество аргументов
    от номера: указатель стэка – ( количество аргументов – 1 )
    из объекта: активный контекст
    в номер: 0
    в объект: ряд аргументов.
  сам вытолкнуть: количество аргументов.
  сам протолкнуть: сообщение.
  количество аргументов ← 1.
```

Следующая процедура инициализирует номера используемые для доступа к полям *Сообщения*.

```
инициализировать номера Сообщения
  Номер селектора сообщения ← 0.
  Номер аргументов сообщения ← 1.
  Размер Сообщения ← 2.
```

Поле класса описывающее экземпляр содержит указатель *Малого целого* которое кодирует следующие четыре порции информации:

1. Содержат ли поля экземпляра указатели на объекты или численные значения.

2. Адресуются ли поля экземпляра в словах или в байтах.

3. Содержит ли экземпляр за фиксированными полями нумерованные поля.

4. Количество фиксированных полей у экземпляра.

Рисунок 27.8 показывает эту информацию закодированную в определении экземпляра.

Эти четыре части информации зависят друг от друга. Если поля экземпляра содержат указатели на объекты, то они должны адресоваться в словах. Если поля экземпляра содержат численные значения, то экземпляр будет иметь нумерованные поля и не иметь фиксированных полей.

```
определение экземпляра: указатель класса
  ↑ память
    извлечь указатель: Номер определения экземпляра
    из объекта: указатель класса.

это указатели: указатель класса
  | флаг указатели |
  флаг указатели ← сам
    извлечь биты от: 0
    до: 0
    из: ( сам определение экземпляра: указатель класса ).
  ↑ флаг указатели = 1.

это слова: указатель класса
  | флаг слова |
  флаг слова ← сам
    извлечь биты от: 1
    до: 1
    из: ( сам определение экземпляра: указатель класса ).
  ↑ флаг слова = 1.

это нумерованный: указатель класса
  | флаг нумерованный |
  флаг нумерованный ← сам
    извлечь биты от: 2
    до: 2
    из: ( сам определение экземпляра: указатель класса ).
  ↑ флаг нумерованный = 1.

фиксированных полей: указатель класса
  ↑ сам
    извлечь биты от: 4
```

до: 14
из: (сам определение экземпляра: указатель класса).

Замечание: определение экземпляра для *Откомпилированного метода* не описывает точно структуру иго экземпляров т.к. *Откомпилированные методы* не однородны. Определение экземпляра указывает что экземпляры не содержат указателей и адресация к полям побайтная. Это верно только для части байткодов. Память объектов должна знать что *Откомпилированные методы* являются особым случаем и что в действительности они содержат несколько указателей. Для всех других классов определение экземпляра точно.

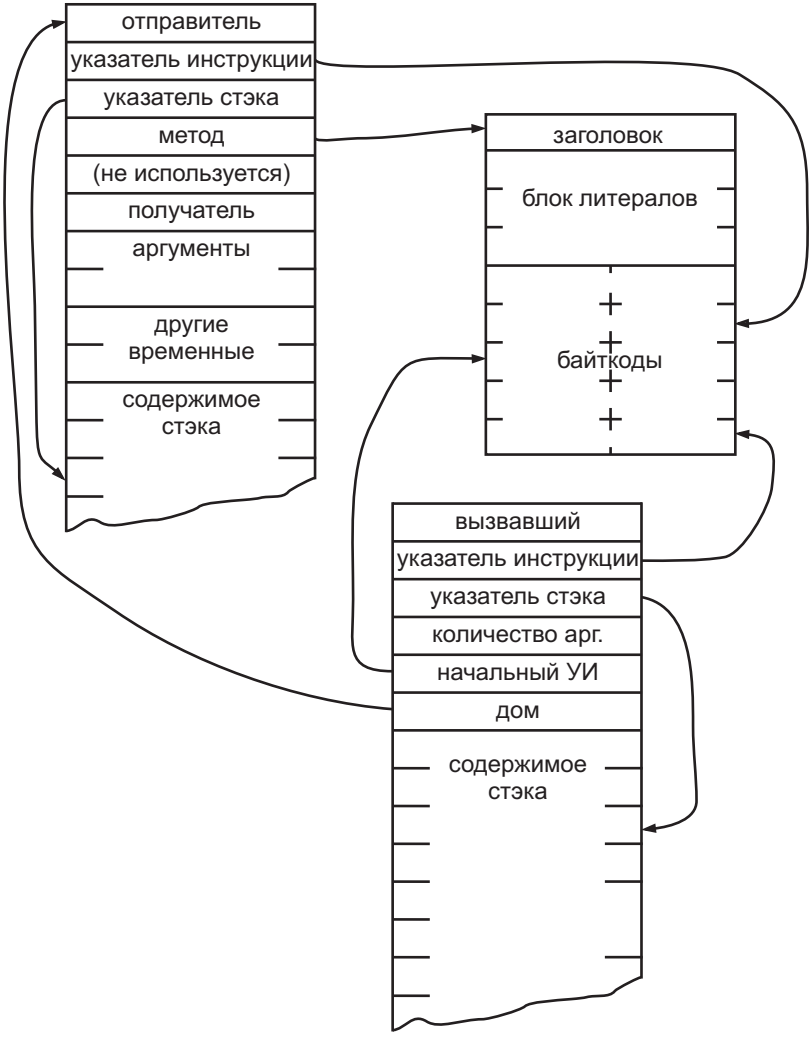

отправитель
указатель инструкции
указатель стэка
метод
(не используется)
получатель
аргументы
другие временные
содержимое стэка
заголовок
блок литералов
байткоды
вызвавший
указатель инструкции
указатель стэка
количество арг.
начальный УИ
дом
содержимое стэка

Рис. 27.6

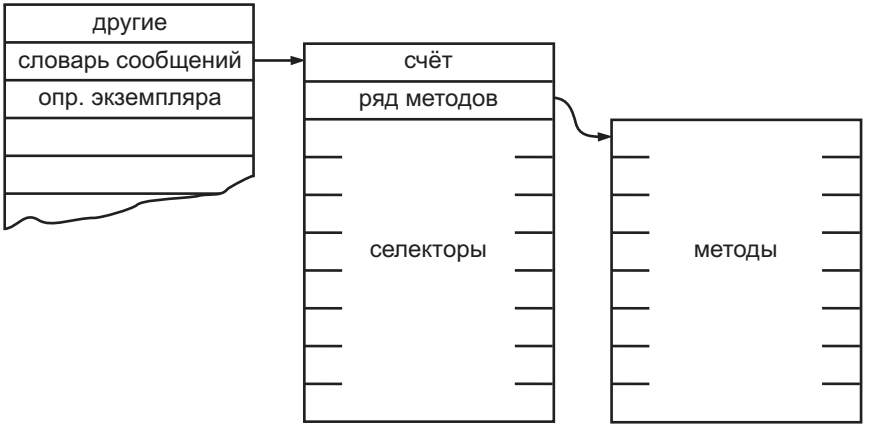


Рис. 27.7

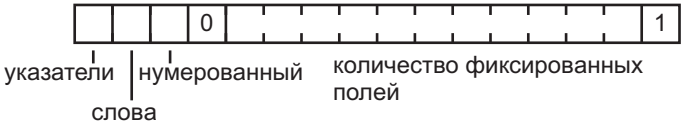


Рис. 27.8

Глава 28

Формальное определение интерпретатора

Оглавление

Главный цикл интерпретатора Смолтока последовательно извлекает байткоды из *Откомпилированного метода* и переходит к процедуре которая выполняет операции указанные байткодом. Процедура *достать байт* достаёт байт на который ссылается указатель инструкции активного контекста и увеличивает указатель инструкции.

```
извлечь байт
  | байт |
  байт ← память извлечь байт: указатель инструкции из объекта: метод.
  указатель инструкции ← указатель инструкции + 1.
  ↑ байт.
```

Т.к. переключение процессов возможно только между байткодами, то первое действие, выполняемое интерпретатором в главном

цикле, это вызов процедуры которая, при необходимости, переключает процессы. Процедура *проверить переключение процессов* будет описана в следующей главе вместе с процедурами элементарных методов планирующих процессы. После проверки переключения процессов извлекается байткод (возможно из нового процесса) и происходит переход к соответствующей процедуре.

```
интерпретировать
    [истина.] пока истина: [сам цикл.].

цикл
    сам проверить переключение процессов.
    текущий байткод ⟵ сам извлечь байт.
    сам выполнить этот байткод.
```

Таблица на странице 486 перечисляет байткоды Смолтока. Байткоды указаны промежутками с подобными функциями. Например, первый промежуток байткодов от 0 до 15 показан ниже.

0-15	0000нннн	Поместить переменную экземпляра 0000нннн.

Каждый промежуток байткодов сопровождается образцом битов и комментарием о функции байткодов. Образец битов показывает двоичное представление промежутка байткодов. Нули и единицы используются для битов одинаковых для всех байткодов диапазона. Т.к. все числа от 0 до 15 имеют четыре нуля в старших битах, то эти биты показаны как 0000. Для битов изменяющихся в данном промежутке используются маленькие буквы. Значение каждой буквы может быть либо 0 либо 1. Буквы используемые в образце могут быть включены в комментарий чтобы сослаться на значение этих битов. Комментарий для первого промежутка байткодов указывает что младшие четыре бита байткода задают номер переменной экземпляра помещаемой на стэк.

Байткоды Смолтока

Промежуток	*Биты*	*Функция*
0-15	0000нннн	Поместить переменную экземпляра 0000нннн.

16-31	0001нннн	Поместить временную переменную 0001нннн.
32-63	001ннннн	Поместить литерал константу 001ннннн.
64-95	010ннннн	Поместить литерал переменную 010ннннн.
96-103	01100ннн	Извлечь и запомнить в переменную экземпляра 01100ннн.
104-111	01101ннн	Извлечь и запомнить во временную переменную 01101ннн.
112-119	01110ннн	Поместить (получателя, истину, ложь, пусто, -1, 0, 1, 2) [ннн].
120-123	011110нн	Вернуть из сообщения (получателя, истину, ложь, пусто) [нн].
124-125	0111110н	Вернуть вершину стэка из (сообщения, блока) [н].
126-127	0111111н	Не используются
128	10000000 ппрррррр	Поместить (переменную получателя, временную переменную, константу литерал, переменную литерал) [пп] №рррррр.
129	10000001 ппрррррр	Сохранить в (переменную получателя, временную переменную, константу литерал, переменную литерал) [пп] №рррррр.
130	10000010 ппрррррр	Снять и сохранить в (переменную получателя, временную переменную, константу литерал, переменную литерал) [пп] №рррррр.
131	10000011 пппрррррр	Послать селектор литерал №рррррр с количеством аргументов ппп.
132	10000100 пппппппп рррррррр	Послать селектор литерал №рррррррр с количеством аргументов пппппппп.
133	10000101 пппрррррр	Послать надклассу селектор №рррррр с количеством аргументов ппп.
134	10000110 пппппппп рррррррр	Послать надклассу селектор №рррррррр с количеством аргументов пппппппп.
135	10000111	Снять вершину стэка.
136	10001000	Удвоить вершину стэка.

137	10001001	Поместить активный контекст.
138-143		Не используются
144-151	10010ннн	Прыгнуть через ннн+1 (т.е. от 1 до 8).
152-159	10011ннн	Снять и прыгнуть если Ложь через ннн+1 (т.е. от 1 до 8).
160-167	10100ннн пппппппп	Прыгнуть через (ннн-4)*256 + пппппппп
168-171	101010нн пппппппп	Снять и прыгнуть если Истина через нн*256 + пппппппп.
172-175	101011нн пппппппп	Снять и прыгнуть если Ложь через нн*256 + пппппппп.
176-191	1011нннн	Послать арифметическое сообщение №нннн.
192-207	1100нннн	Послать специальное сообщение №нннн.
208-223	1101нннн	Послать селектор литерал №нннн без аргументов.
224-239	1110нннн	Послать селектор литерал №нннн с одним аргументом.
240-255	1111нннн	Послать селектор литерал №нннн с двумя аргументами.

Также переменные биты из шаблона иногда используются в комментарии как номера относительно нуля. Например:

120-123	011110нн	Вернуть из сообщения (получателя, истину, ложь, пусто) [нн].

указывает что байткод 120 возвращает получателя, байткод 121 возвращает *истину*, байткод 122 возвращает *ложъ* а байткод 123 возвращает *пусто*.

Диапазон байткодов которые имеют расширение включает больше одного образца. Например:

131	10000011 пппрррррр	Послать селектор литерал №рррррр с количеством аргументов ппп.

Есть четыре основных типа байткодов.

- *байткоды стэка* перемещают объекты между памятью объектов и стэком выполнения активного контекста. В них входят

и байткоды помещения и байткоды запоминания описанные в Главе 26.

- *байткоды прыжков* изменяют указатель инструкции активного контекста.
- *байткоды посылки* вызывают *Откомпилированный метод* или элементарный метод.
- *байткоды возврата* заканчивают выполнение *Откомпилированного метода.*

Не все байткоды одного типа идут подряд, поэтому главная процедура выполнения байткодов имеет семь разделов каждый из которых вызывает одну из четырёх процедур (*байткод стэка*, *байткод прыжка*, *байткод посылки* или *байткод возврата*). Эти четыре процедуры будут описаны в следующих четырёх подразделах.

выполнить этот байткод
(текущий байткод между: 0 и: 119) истина: [↑ сам байткод стэка.].
(текущий байткод между: 120 и: 127) истина: [↑ сам байткод возврата.].
(текущий байткод между: 128 и: 130) истина: [↑ сам байткод стэка.].
(текущий байткод между: 131 и: 134) истина: [↑ сам байткод посылки.].
(текущий байткод между: 135 и: 137) истина: [↑ сам байткод стэка.].
(текущий байткод между: 144 и: 175) истина: [↑ сам байткод прыжка.].
(текущий байткод между: 176 и: 255) истина: [↑ сам байткод посылки.].

Байткоды 176-191 ссылаются на арифметические сообщения. Этими сообщениями являются: +, -, <, >, <=, >=, =, ~=, *, /, \\, @, *сдвинуть биты:*, //, *побитовое и:*, *побитовое или:*

Байткоды 192-207 ссылаются на специальные сообщения. Этими сообщениями являются: *от:**, *от:пом:**, *размер**, *следующий**, *пом следующим:**, *в конце**, ==, *класс*, *экземпляр блока:*, *значение*, *значение:*, *делать:**, *новый**, *новый:**, *икс**, *игрек**.

Селекторы помеченные звёздочкой могут быть изменены с помощью модификации компилятора.

28.1 Байткоды стэка

Все байткоды стэка производят простые операции над стэком выполнения активного контекста.

- 107 байткодов помещают на стэк указатель объекта
  - 99 помещают указатель объекта находящегося в памяти объектов
  - 7 помещают указатель на объект константу
  - 1 помещает регистр интерпретатора активный контекст
- 18 байткодов сохраняют указатель объекта находящийся на стэке в память объектов
  - 17 из них также удаляют этот объект с вершины стэка
  - 1 оставляет этот объект на стэке
- 1 байткод удаляет со стэка указатель объекта никуда его не сохраняя

Процедуры используемые для манипуляций стэком были описаны в предыдущей главе в разделе о контекстах (*протолкнуть:*, *вытолкнуть стэк*, *вытолкнуть:*). Процедура *байткод стэка* вызывает соответствующую процедуру для текущего байткода.

байткод стэка

```
( текущий байткод между: 0 и: 15 )
  истина: [↑сам байткод поместить переменную получателя.].
( текущий байткод между: 16 и: 31 )
  истина: [↑сам байткод поместить временную переменную.].
( текущий байткод между: 32 и: 63 )
  истина: [↑сам байткод помесить литерал константу.].
( текущий байткод между: 64 и: 95 )
  истина: [↑сам байткод поместить литерал переменную.].
( текущий байткод между: 96 и: 103 )
  истина: [↑сам байткод сохранить в переменную экземпляра и вытолкнуть.].
( текущий байткод между: 104 и: 111 )
```

```
    истина: [↑сам байткод сохранить во временную переменную и вытолкнуть.].
  текущий байткод = 112 истина: [↑сам байткод поместить получателя.].
  (текущий байткод между: 113 и: 119)
    истина: [↑сам байткод поместить константу.].
  текущий байткод = 128 истина: [↑сам байткод расширенное помещение.].
  текущий байткод = 129 истина: [↑сам расширенный байткод сохранить.].
  текущий байткод = 130
    истина: [↑сам расширенный байткод сохранить и вытолкнуть.].
  текущий байткод = 135 истина: [↑сам байткод вытолкнуть стэк.].
  текущий байткод = 136 истина: [↑сам байткод удвоить вершину.].
  текущий байткод = 137 истина: [↑сам батйкод поместить активный контекст.].
```

Есть однобайтные инструкции помещающие на стэк 16 первых переменных экземпляра и 16 первых переменных блока временных. Вспомните что в блок временных входят аргументы и временные переменные.

```
байткод поместить переменную получателя
  | номер поля |
  номер поля ← сам извлечь биты от: 12 до: 15 из: текущий байткод.
  сам поместить переменную получателя: номер поля.

поместить переменную получателя: номер поля
  сам
    протолкнуть: (память извлечь указатель: номер поля из объекта: получатель).

байткод поместить временную переменную
  | номер поля |
  номер поля ← сам извлечь биты от: 12 до: 15 из: текущий байткод.
  сам поместить временную переменную: номер поля.

поместить временную переменную: номер временной
  сам протолкнуть: (сам временная: номер временной).
```

Также есть однобайтные инструкции которые ссылаются на 32 первых положения в блоке литералов контекста активного метода. Содержимое одного из этих положений может быть помещено процедурой *байткод поместить литерал константу*. Содержимое поля *Ассоциации значение* хранящееся в одном из этих положений может быть помещено на стэк процедурой *байткод поместить литерал переменную*.

байткод поместить литерал константу
| номер поля |
номер поля ← сам извлечь биты от: 11 до: 15 из: текущий байткод.
сам поместить литерал константу: номер поля.

поместить литерал константу: номер литерала
сам протолкнуть: (сам литерал: номер литерала).

байткод поместить литерал переменную
| номер поля |
номер поля ← сам извлечь биты от: 11 до: 15 из: текущий байткод.
сам поместить литерал переменную: номер поля.

поместить литерал переменную: номер литерала
| ассоциация |
ассоциация ← сам литерал: номер литерала.
сам
протолкнуть: (память извлечь указатель: Номер значения из объекта: ассоциация).

Ассоциации это объекты с двумя полями, одно из них для имени а другое для значения. *Ассоциации* используются для реализации разделяемых переменных (глобальных переменных, переменных класса и переменных пула). Следующая процедура инициализирует номер используемый для извлечения из *Ассоциации* значения поля.

инициализировать номер Ассоциации
Номер значения ← 1.

Расширенные байткоды помещения могут выполнять любую из вышеописанных четырёх операций (переменные получателя, поло-

жения блока временных, литералы константы или литералы переменные). Однако вместо предела в 16 или 32 доступных переменных, они могут обращаться до 64-х переменных экземпляра, положений блока временных, литералов констант или литералов переменных. Расширение байткода помещение идёт за байтом чьи два старших бита задают тип помещения и чьи младшие шесть битов задают используемое смещение.

байткод расширенное помещение

```
| описатель тип переменной номер переменной |
описатель ← сам извлечь байт.
тип переменной ← сам извлечь биты от: 8 до: 9 из: описатель.
номер переменной ← сам извлечь биты от: 10 до: 15 из: описатель.
тип переменной = 0
    истина: [↑сам поместить переменную получателя: номер переменной.].
тип переменной = 1
    истина: [↑сам поместить временную переменную: номер переменной.].
тип переменной = 2
    истина: [↑сам поместить литерал константу: номер переменной.].
тип переменной = 3
    истина: [↑сам поместить литерал переменную: номер переменной.].
```

Процедура *байткод поместить получателя* помещает на стэк указатель на получателя активного контекста. Это соответствует использованию в методах *себя* или *нада*.

байткод поместить получателя

```
сам протолкнуть: получатель.
```

Процедура *байткод удвоить вершину* помещает на стэк копию верхнего указателя находящегося на стэке.

байткод удвоить вершину

```
↑сам протолкнуть: сам вершина стэка.
```

Процедура *байткод поместить константу* помещает на стэк указатель на одну из семи констант (*истина*, *ложь*, *пусто*, -1, 0, 1 или 2).

байткод поместить константу

```
  текущий байткод = 113 истина: [↑сам протолкнуть: Указатель на истину.].
  текущий байткод = 114 истина: [↑сам протолкнуть: Указатель на ложь.].
  текущий байткод = 115 истина: [↑сам протолкнуть: Указатель на пусто.].
   текущий байткод = 116
      истина: [↑сам протолкнуть: Указатель на минус еденицу.].
  текущий байткод = 117 истина: [↑сам протолкнуть: Указатель на ноль.].
  текущий байткод = 118 истина: [↑сам протолкнуть: Указатель на еденицу.].
  текущий байткод = 119 истина: [↑сам протолкнуть: Указатель на двойку.].
```

Процедура *байткод поместить активный контекст* помещает на стэк указатель на активный контекст. Это соответствует использованию в методах *этого контекста*.

батйкод поместить активный контекст

```
  сам протолкнуть: активный контекст.
```

Байткоды сохранения перемещают ссылки в другом направлении по сравнению с байткодами помещения; с вершины стэка в переменные экземпляра получателя, временные переменные или блок литералов. Есть однобайтная версия сохраняющая указатель в первые восемь переменных получателя или в блок временных и затем удаляющая со стэка этот указатель.

байткод сохранить в переменную экземпляра и вытолкнуть

```
  | номер переменной |
   номер переменной ← сам извлечь биты от: 13 до: 15 из: текущий байткод.
  память
     сохранить указатель: номер переменной
     в объект: получатель
     со значением: сам вытолкнуть стэк.
```

байткод сохранить во временную переменную и вытолкнуть

```
  | номер переменной |
```

```
    номер переменной ← сам извлечь биты от: 13 до: 15 из: текущий байткод.
  память
    сохранить указатель: номер переменной + Начало блока временных
      в объект: домашний контекст
      со значением: сам вытолкнуть стэк.
```

Сохранение в переменные отличные от доступных однобйтной версии осуществляется двумя расширенными байткодами сохранения. Один из них выталкивает стэк после сохранения а другой нет. У обоих расширенных сохранений следующий байт имеет ту же форму что и для расширенного помещения. Однако не допускается расширенный байткод сохранения с байтом вида 10xxxxxx, т.к. это бы означало изменение значение литерала константы.

расширенный байткод сохранить и вытолкнуть

```
  сам расширенный байткод сохранить.
  сам байткод вытолкнуть стэк.
```

расширенный байткод сохранить

```
  | описатель тип переменной номер переменной ассоциация |
  описатель ← сам извлечь байт.
  тип переменной ← сам извлечь биты от: 8 до: 9 из: описатель.
  номер переменной ← сам извлечь биты от: 10 до: 15 из: описатель.
  тип переменной = 0
    истина: [
      ↑ память
        сохранить указатель: номер переменной
        в объект: получатель
        со значением: сам вершина стэка. ].
  тип переменной = 1
    истина: [
      ↑ память
       сохранить указатель: номер переменной + Начало блока временных
        в объект: домашний контекст
        со значением: сам вершина стэка. ].
```

```
тип переменной = 2 истина: [↑сам ошибка: 'illegal store'. ].
тип переменной = 3
  истина: [
    ассоциация ← сам литерал: номер переменной.
    ↑память
      сохранить указатель: Номер значения
      в объект: ассоциация
      со значением: сам вершина стэка. ].
```

Последний байткод стэка просто удаляет со стэка верхний объект.

байткод вытолкнуть стэк

```
сам вытолкнуть стэк.
```

28.2 Байткоды прыжков

Байткоды прыжков изменяют указатель инструкции активного контекста на заданную величину. Безусловные прыжки изменяют указатель инструкции когда они встречаются. Условные прыжки изменяют указатель инструкции если указатель объекта на вершине стэка является заданным *Логическим* объектом (*истиной* или *ложью*). И безусловные и условные прыжки имеют короткую (однобайтную) и длинную (двухбайтную) форму.

байткод прыжка

```
( текущий байткод между: 144 и: 151 )
    истина: [↑сам короткий безусловный прыжок. ].
( текущий байткод между: 152 и: 159 ) истина: [↑сам короткий условный прыжок. ].
( текущий байткод между: 160 и: 167 )
    истина: [↑сам длинный безусловный прыжок. ].
( текущий байткод между: 168 и: 175 ) истина: [↑сам длинный условный прыжок. ].
```

Байткоды прыжков для изменения номера байткода используют процедуру *прыгнуть:*.

прыгнуть: смещение

```
указатель инструкции ← указатель инструкции + смещение.
```

Восемь коротких безусловных прыжков увеличивают указатель инструкции на величину от 1 до 8.

короткий безусловный прыжок

```
| смещение |
смещение ← сам извлечь биты от: 13 до: 15 из: текущий байткод.
сам прыгнуть: смещение + 1.
```

За восемью длинными безусловнми байткодами следует другой байт. Три младших бита байткода прыжка представляют три старших бита 11-битной величины, в виде дополнения до двух со смещением, которая добавляется к указателю инструкции. Следующий байт является восемью младшими битами этого смещения. Поэтому длинный безусловный прыжок может прыгать на величину до 1023 байт вперёд и до 1024 байт назад.

длинный безусловный прыжок

```
| смещение |
смещение ← сам извлечь биты от: 13 до: 15 из: текущий байткод.
сам прыгнуть: смещение - 4 * 256 + сам извлечь байт.
```

Условные прыжки используют процедуру *прыгнуть если:на:* для проверки вершины стэка и решения нужно ли прыгать. После проверки вершина стэка удаляется.

прыгнуть если: условие на: смещение

```
| логическое |
логическое ← сам вытолкнуть стэк.
логическое = условие
    истина: [сам прыгнуть: смещение.]
    ложь: [
    логическое = Указатель на истину | (логическое = Указатель на ложь)
        ложь: [
            сам отменить выталкивание: 1.
            сам послать должен быть логическим.].].
```

Условные прыжки используются в откомпилированной форме сообщений к логическим значениям (т.е. сообщениями *истина:* и *пока истина:*). Если во время условного прыжка вершина стэка это не *истина* или *ложь*, то это является ошибкой т.к. отличному от логического объекту послано сообщение которое понимают только

логические значения. Вместо посылки сообщения *не понимаю:* интерпретатор посылает себе сообщение *должен быть логическим.*

```
послать должен быть логическим
    сам
        послать селектор: Селектор должен быть логическим
        количество аргументов: 0.
```

Процедура *послать селектор:количество аргументов:* описывается в следующем разделе о байткодах посылки.

Восемь коротких условных прыжков увеличивают указатель инструкции на величину от 1 до 8 если вершина стэка это *ложь*.

```
короткий условный прыжок
    | смещение |
    смещение ← сам извлечь биты от: 13 до: 15 из: текущий байткод.
    сам прыгнуть если: Указатель на ложь на: смещение + 1.
```

Поэтому возможны три исхода короткого условного прыжка:

- Если вершина стэка это *ложь*, то происходт прыжок.
- Если вершина стэка это *истина*, то прыжка не происходит.
- Если вершиной стэка не является ни *ложью* ни *истиной*, то посылается сообщение *должен быть логическим.*

Половина условных длинных прыжков выполняется если вершина стэка это *ложь*, а другая половина выполняется если вершина это *истина*. Два младших бита байткода становятся старшими битами 10-битного беззнакового смещения. Байт следующий за байткодом прыжка представляет младшие восемь битов смещения. Поэтому условные прыжки могут прыгать на величину до 1023 байтов вперёд.

```
длинный условный прыжок
    | смещение |
    смещение ← сам извлечь биты от: 14 до: 15 из: текущий байткод.
    смещение ← смещение * 256 + сам извлечь байт.
    ( текущий байткод между: 168 и: 171 )
        истина: [↑ сам прыгнуть если: Указатель на истину на: смещение. ].
```

```
(текущий байткод между: 172 и: 175)
  истина: [↑сам прыгнуть если: Указатель на ложь на: смещение.].
```

28.3 Байткоды посылки

Байткоды посылки заставляют сообщения посылаться. Указатели объекта получателя и аргументов сообщения находятся на стэке выполнения активного контекста. Байткод посылки задаёт селектор сообщения и количество аргументов берущихся со стэка. Также количество аргументов указывается *Откомпилированным методом* вызываемым сообщением. Компилятор гарантирует что это информация излишняя за исключением случая когда *Откомпилированный метод* вызывается сообщением *выполнить:*, в этом случае проверяется что *Откомпилированный метод* принимает правильное количество аргументов. Сообщение *выполнить:* будет обсуждаться в следующей главе в разделе о управляющих элементарных методах.

Селекторы большинства сообщений находятся в блоке литералов *Откомпилированного метода*. Байткоды с селектором литералом и расширенные байткоды посылки задают количество аргументов сообщения и номер селектора в блоке литералов. 32 байткода для специальных селекторов задают номер селектора в *Ряде* из памяти объектов и количество аргументов. Этот *Ряд* разделяется всеми *Откомпилированными методами* системы.

байткод посылки

```
(текущий байткод между: 131 и: 134)
  истина: [↑сам расширенный байткод посылки.].
(текущий байткод между: 176 и: 207)
  истина: [↑сам байткод послать специальный селектор.].
(текущий байткод между: 208 и: 255)
  истина: [↑сам байткод послать селектор литерал.].
```

Байткоды с селектором литералом это один байт который может задавать 0, 1 или 2 аргумента и селектор в любом из 16 положений блока литералов. И номер селектора и количество аргументов кодируются битами байткода.

```
байткод послать селектор литерал
  | селектор |
  селектор ← сам литерал: ( сам извлечь биты от: 12 до: 15 из: текущий байткод ).
  сам
    послать селектор: селектор
      количество аргументов: ( сам извлечь биты от: 10 до: 11 из: текущий байткод ) — 1.
```

Большинство байткодов посылки, после нахождения соответствующего селектора и количества аргументов, вызывают процедуру *послать селектор:количество аргументов:*. Эта процедура устанавливает регистры *селектор сообщения* и *количество аргументов* которые доступны другим процедурам интерпретатора которые будут искать сообщение и возможно активировать метод.

```
послать селектор: селектор количество аргументов: количество
  | новый получатель |
  селектор сообщения ← селектор.
  количество аргументов ← количество.
  новый получатель ← сам значение стэка: количество аргументов.
  сам послать селектор классу: ( память извлечь класс: новый получатель ).

послать селектор классу: указатель класса
  сам найти новый метод в классе: указатель класса.
  сам выполнить новый метод.
```

Интерпретатор использует кэш методов чтобы уменьшить количество необходимых поисков в словаре при нахождении *Откомпилированного метода* связанного с селектором. Кэш методов можно не использовать заменив в процедуре *послать селектор классу:* вызов *искать метод в классе:* на вызов *найти новый метод в классе:*. Процедура *искать метод в классе:* описана в предыдущей главе в разделе о классах. Кэш может быть реализован различными способами. Следующая процедура использует для каждой записи четыре последовательных положения в *Ряде*. Четыре положения

хранят селектор, класс, *Откомпилированный метод* и номер элементарного метода. Эта процедура не позволяет reprobes.

```
найти новый метод в классе: класс
  | хэш |
  хэш ← ( ( ( селектор сообщения побитовое и: класс ) побитовое и:
16oEE ) сдвинуть биты: 2 ) + 1.
  ( ( кэш методов от: хэш ) = селектор сообщения
    и: [ ( кэш методов от: хэш + 1 ) = класс. ] )
      истина: [
        новый метод ← кэш методов от: хэш + 2.
        номер элементарного метода ← кэш методов от: хэш + 3. ]
      ложь: [
        сам искать метод в классе: класс.
        кэш методов от: хэш пом: селектор сообщения.
        кэш методов от: хэш + 1 пом: класс.
        кэш методов от: хэш + 2 пом: новый метод.
       кэш методов от: хэш + 3 пом: номер элементарного метода. ].
```

Кэш методов инициализируется следующей процедурой.

```
инициализировать кэш методов
  размер кэша методов ← 1024.
  кэш методов ← Ряд новый: размер кэша методов.
```

Процедура *выполнить новый метод* вызывает элементарную процедура если она связана с *Откомпилированным методом*. Процедура *ответ элементарного метода* возвращает *ложь* если элементарного метода не указано или элементарный метод не может произвести результат. В этом случае активируется *Откомпилированный метод*. Элементарные методы и процедура *ответ элементарного метода* будут описаны в следующей главе.

```
выполнить новый метод
    сам ответ элементарного метода ложь: [сам активировать новый метод. ].
```

Процедура активирующая метод создаёт *Контекст метода* и передаёт получателя и аргументы из стэка текущего активного контекста в стэк нового контекста. Затем этот новый контекст становится активным контекстом интерпретатора.

```
активировать новый метод
  | размер контекста новый контекст новый получатель |
  ( сам флаг большого контекста: новый метод ) = 1
    истина: [ размер контекста ← 32 + Начало блока временных. ]
    ложь: [ размер контекста ← 12 + Начало блока временных. ].
  новый контекст ← память
    экземпляр класса: Указатель на класс Контекст метода
    с указателями: размер контекста.
  память
    сохранить указатель: Номер отправителя
    в объект: новый контекст
    со значением: активный контекст.
  сам
   сохранить значение указателя инструкции: ( сам начальный указатель инструкции метода: новый метод )
    в контекст: новый контекст.
  сам
   сохранить значение указателя стэка: ( сам количество временных: новый метод )
    в контекст: новый контекст.
  память
    сохранить указатель: Номер метода
    в объект: новый контекст
    со значением: новый метод.
  сам
    перенести: количество аргументов + 1
    от номера: указатель стэка − количество аргументов
    из объекта: активный контекст
    в номер: Номер получателя
    в объект: новый контекст.
  сам вытолкнуть: количество аргументов + 1.
  сам новый активный контекст: новый контекст.
```

Есть четыре расширенных байткода посылки. Первые два имеют тот же эффект что и байткоды с селектором литералом за исключением того что номер селектора и количество аргументов находятся в одном из двух следующих байтов а не в самом байткоде. Другие

два типа расширенных байткодов используются для сообщений надклассу.

расширенный байткод посылки

```
текущий байткод = 131 истина: [↑сам байткод посылки с одним расширением.].
текущий байткод = 132 истина: [↑сам байткод посылки с двумя расширениями.].
текущий байткод = 133
    истина: [↑сам байткод посылки наду с одним расширением.].
текущий байткод = 134
    истина: [↑сам байткод посылки наду с двумя расширениями.].
```

Первый вид расширенного байткода посылки использует один байт который задаёт количество аргументов в своих трёх старших битах и номер селектора в пяти младших битах.

байткод посылки с одним расширением

```
| описатель номер селектора |
описатель ← сам извлечь байт.
номер селектора ← сам извлечь биты от: 11 до: 15 из: описатель.
сам
    послать селектор: (сам литерал: номер селектора)
        количество аргументов: (сам извлечь биты от: 8 до: 10 из: описатель).
```

Второй вид расширенного байткода посылки использует два байта; первый это количество аргументов а второй это номер селектора в блоке литералов.

байткод посылки с двумя расширениями

```
| количество селектор |
количество ← сам извлечь байт.
селектор ← сам литерал: сам извлечь байт.
сам послать селектор: селектор количество аргументов: количество.
```

Когда компилятор встречает в методе сообщение *наду*, то он использует байткод который помещает на стэк в качестве получателя *себя*, но в место обычного байткода посылки использует для указания селектора расширенный байткод посылки наду. Два расширен-

ных байткода посылки наду подобны двум расширенным байткодам посылки. За первым следует один байт а за вторым два байта которые интерпретируются в точности так же как и для расширенных байткодов посылки. Единственная разница в действиях этих байткодов это то что они начинают поиск метода в надклассе класса в котором находится текущий *Откомпилированный метод*. Заметьте что это не обязательно непосредственный надкласс *себя*. В частности, этот класс не будет непосредственным надклассом *себя* если *Откомпилированный метод* содержащий расширенный байткод посылки наду находится в надклассе *себя*. Все *Откомпилированные методы*, содержащие расширенные байткоды посылки наду, содержат в своём блоке литералов, в качестве последнего литерала переменной, класс в котором они находятся.

байткод посылки наду с одним расширением

```
| описатель номер селектора класс метода |
описатель ← сам извлечь байт.
количество аргументов ← сам извлечь биты от: 8 до: 10 из: описатель.
номер селектора ← сам извлечь биты от: 11 до: 15 из: описатель.
селектор сообщения ← сам литерал: номер селектора.
класс метода ← сам класс метода: метод.
сам послать селектор классу: ( сам надкласс для: класс метода ).
```

байткод посылки наду с двумя расширениями

```
| класс метода |
количество аргументов ← сам извлечь байт.
селектор сообщения ← сам литерал: сам извлечь байт.
класс метода ← сам класс метода: метод.
сам послать селектор классу: ( сам надкласс для: класс метода ).
```

Набор специальных селекторов может быть использован в сообщении без помещения их в блок литералов. *Ряд* в памяти объектов содержит указатели на селекторы в чередующемся порядке. Количество аргументов каждого селектора хранится в положении за самим указателем селектора. Процедура *ответ элементарного метода специального селектора* будет описана в следующей главе.

байткод послать специальный селектор

```
| номер селектора селектор количество |
сам ответ элементарного метода специального селектора
  ложь: [
    номер селектора ← текущий байткод – 176 * 2.
    селектор ← память
      извлечь указатель: номер селектора
      из объекта: Указатель на специальные селекторы.
    количество ← сам
      достать целое: номер селектора + 1
      из объекта: Указатель на специальные селекторы.
    сам послать селектор: селектор количество аргументов: количество. ].
```

28.4 Байткоды возврата

Есть шесть байткодов возвращающих управление и значение из контекста; пять возвращают значение сообщения (явно вызванных "↑"или неявно в конце метода) и один для возврата значения блока (вызывается неявно в конце блока). Разница между этими двумя типами возврата в том что первый тип возвращает управление отправителю домашнего контекста а второй тип возвращает управление вызвавшему активный контекст. Значения возвращаемое пятью байткодами это: получатель (*сам*), *истина*, *ложь*, *пусто* или вершина стэка. Последний байткод возвращает в качестве значения блока вершину стэка.

байткод возврата

```
текущий байткод = 120
    истина: [↑сам вернуть значение: получатель к: сам отправитель. ].
текущий байткод = 121
    истина: [↑сам вернуть значение: Указатель на истину к: сам отправитель. ].
текущий байткод = 122
    истина: [↑сам вернуть значение: Указатель на ложь к: сам отправитель. ].
текущий байткод = 123
```

```
        истина: [↑сам вернуть значение: Указатель на пусто к: сам отправитель.].
    текущий байткод = 124
        истина: [↑сам вернуть значение: сам вытолкнуть стэк к: сам отправитель.].
    текущий байткод = 125
        истина: [↑сам вернуть значение: сам вытолкнуть стэк к: сам вызвавший.].
```

Простым способом возвращения значения контексту было бы просто сделать его активным контекстом и поместить на его стэк значение.

```
просто вернуть значение: указатель результата к: указатель контекста
    сам новый активный контекст: указатель контекста.
    сам протолкнуть: указатель результата.
```

Однако есть три ситуации в которых эта процедура слишком проста чтобы работать правильно. Если отправитель активного контекста будет *пусто* то эта процедура поместить *пусто* в указатель интерпретатора активный контекст, переведя систему к неприятному останову. Чтобы избежать этого, настоящая процедура *вернуть значение:к:* сначала проверяет *пусто* ли отправитель. Интерпретатор также предотвращает возврат в контекст из которого уже был произведён возврат. Это делается путём помещения *пусто* в указатель инструкции активного контекста при возврате и проверки на *пусто* указателя инструкции контекста в который происходит возврат. Обе эти ситуации могут возникнуть т.к. контексты это объекты и ими может манипулировать программы пользователя также как и интерпретатор. Если такая ситуация происходит, то интерпретатор посылает активному контексту сообщение информирующее о проблеме. Третья ситуация может возникнуть в системах которые автоматически освобождают объекты на основе подсчёта ссылок. Активный контекст может быть освобождён после возвращения из него. Он, в свою очередь, может содержать только ссылки на только что возвращённый результат. В этом случае результат будет освобождён до того как он будет помещён на стэк нового контекста. Из за этого процедура *вернуть значение:* должна быть более сложной.

```
вернуть значение: указатель результата к: указатель контекста
  | уи отправителя |
  указатель контекста = Указатель на пусто
    истина: [
      сам протолкнуть: активный контекст.
      сам протолкнуть: указатель результата.
      ↑ сам
        послать селектор: Селектор невозможно вернуть
        количество аргументов: 1. ].
  уи отправителя ← память
    извлечь указатель: Номер указателя инструкции
    из объекта: указатель контекста.
  уи отправителя = Указатель на пусто
    истина: [
      сам протолкнуть: активный контекст.
      сам протолкнуть: указатель результата.
      ↑ сам
        послать селектор: Селектор невозможно вернуть
        количество аргументов: 1. ].
  память увеличить ссылки на: указатель результата.
  сам вернуться в активный контекст: указатель контекста.
  сам протолкнуть: указатель результата.
  память уменьшить ссылки на: указатель результата.
```

Эта процедура предотвращает освобождение возвращаемого результата путём увеличения количества ссылок до тех пор пока результат не помещён на новый стэк. Также она помещает результат до переключения активного контекста. Процедура *вернуться в активный контекст:* просто такая же что и процедура *новый активный контекст:* за исключением того что она восстанавливает все кэшированные поля контекста в который происходит возврат, она помещает *пусто* в поля отправитель и указатель инструкции.

```
вернуться в активный контекст: контекст
  память увеличить ссылки на: контекст.
  сам очистить поля контекста.
  память уменьшить ссылки на: активный контекст.
```

```
  активный контекст ⟵ контекст.
  сам извлечь регистры контекста.

очистить поля контекста
  память
    сохранить указатель: Номер отправителя
    в объект: активный контекст
    со значением: Указатель на пусто.
  память
    сохранить указатель: Номер указателя инструкции
    в объект: активный контекст
    со значением: Указатель на пусто.
```

Глава 29

Формальные определения элементарных методов

Оглавление

Обычно при посылке сообщения интерпретатор отвечает путём выполнения *Откомпилированного метода*. Выполнение состоит из создания нового *Контекста метода* для этого *Откомпилированного метода* и выполнения его байткодов до тех пор пока не встретится байткод возврата. Однако, некоторые сообщения могут возвращать результат элементарно. Ответ элементарного метода осуществляется интерпретатором напрямую, без создания нового контекста и выполнения байткодов. Каждый элементарный ответ интерпретатора описан элементарной процедурой. Элементарная процедура удаляет со стэка получателя сообщения и аргументы и заменяет их на соответствующий результат. Некоторые элементарные процедуры имеют другие эффекты для памяти объектов или

для некоторых устройств. После завершения элементарной процедуры интерпретатор продолжает интерпретацию байткодов находящихся после байткода посылки вызвавшего элементарную процедуру.

В любой момент своего выполнения элементарная процедура может определить что невозможно найти ответ. Это может случиться, например, из за неправильного класса аргумента. Это называется неудачей элементарного метода. При неудаче элементарного метода, будет выполнен метод Смолтока связанный с селектором в классе получателя так как будто элементарного метода не существует.

Следующая таблица показывает пары класс-селектор связанные с элементарными процедурами. Некоторые из этих пар класс-селектор не были рассмотрены раньше в этой книге т.к. они являются частью собственного протокола классов. Чтобы система работала правильно некоторые элементарные процедуры должны соответствовать своим определениям. Другие элементарные процедуры необязательны, просто система будет работать менее эффективно если они всегда будут неудаваться. Необязательные процедуры помечены звёздочкой. Методы Смолтока связанные с необязательными элементарными процедурами должны делать всю работу за них. Методы Смолтока связанные с обязательными элементарными процедурами должны только обрабатывать случаи неудачи элементарного метода.

Элементарные методы Смолтока

Номер	*Пара класс-селектор*
1	Малое целое +
2	Малое целое −
3	Малое целое <
4	Малое целое >
5*	Малое целое <=
6*	Малое целое >=
7	Малое целое =
8*	Малое целое ~=
9	Малое целое *
10*	Малое целое /
11*	Малое целое \\
12	Малое целое //

13* Малое целое **частное:**
14 Малое целое **побитовое и:**
15 Малое целое **побитовое или:**
16 Малое целое **побитовое искл или:**
17 Малое целое **сдвинуть биты:**
18* Число **@**
19
20
21* Целое +
Большое положительное целое +
22* Целое −
Большое положительное целое −
23* Целое <
Большое положительное целое <
24* Целое >
Большое положительное целое >
25* Целое <=
Большое положительное целое <=
26* Целое >=
Большое положительное целое >=
27* Целое =
Большое положительное целое =
28* Целое ~=
Большое положительное целое ~=
29* Целое *
Большое положительное целое *
30* Целое /
Большое положительное целое /
31* Целое \\
Большое положительное целое \\
32* Целое //
Большое положительное целое //
33* Целое **частное:**
Большое положительное целое **частное:**
34* Целое **побитовое и:**
Большое положительное целое **побитовое и:**

```
35*     Целое побитовое или:
        Большое положительное целое побитовое или:
36*     Целое побитовое искл или:
        Большое положительное целое побитовое искл или:
37*     Целое сдвинуть биты:
        Большое положительное целое сдвинуть биты:
38
39
40      Малое целое как плавающее
41      Плавающее +
42      Плавающее -
43      Плавающее <
44      Плавающее >
45*     Плавающее <=
46*     Плавающее >=
47      Плавающее =
48*     Плавающее ~=
49      Плавающее *
50      Плавающее /
51      Плавающее усечь
52*     Плавающее дробная часть
53*     Плавающее экспонента
54*     Плавающее умножить на два в степени:
55
56
57
58
59
60      Большое отрицательное целое цифра от:
        Большое положительное целое цифра от:
        Объект от:
        Объект основной от:
61      Большое отрицательное целое цифра от:пом:
        Большое положительное целое цифра от:пом:
        Объект от:пом:
        Объект основной от:пом:
```

62 Набор ряд **размер**
Большое отрицательное целое **длина цифр**
Большое положительное целое **длина цифр**
Объект **основной размер**
Объект **размер**
Цепь **размер**
63 Цепь **от:**
Цепь **основной от:**
64 Цепь **основной от:пом:**
Цепь **от:пом:**
65* Поток чтения **следующий**
Поток чтения записи **следующий**
66* Поток записи **пом следующим:**
67* Позиционируемый поток **в конце**
68 Откомпилированный метод **объект от:**
69 Откомпилированный метод **объект от:пом:**
70 Поведение **основной новый**
Поведение **новый**
Интервал класс **новый**
71 Поведение **новый:**
Поведение **основной новый:**
72 Объект **становится:**
73 Объект **пер экз от:**
74 Объект **пер экз от:пом:**
75 Объект **как УО**
Объект **хэш**
Символ **хэш**
76 Малое целое **как объект**
Малое целое **asObjectNoFail**
77 Поведение **некоторый экземпляр**
78 Объект **следующий экземпляр**
79 Откомпилированный метод класс **новый метод:-заголовок:**
80* Часть контекста **экземпляр блока:**

```
81      Контекст блока значение:значение:значение:значение:
        Контекст блока значение:значение:значение:
        Контекст блока значение:значение:
        Контекст блока значение:
82      Контекст блока значение с аргументами:
83*     Объект выполнить:с:с:с:
        Объект выполнить:с:с:
        Объект выполнить:с:
        Объект выполнить:
84      Объект выполнить:с аргументами:
85      Семафор сигнал
86      Семафор ждать
87      Процесс возобновить
88      Процесс приостановить
89      Поведение сбросить кэш
90*     Датчик ввода элем точка мыши
        Объект элем точка мыши
91
92      Курсор класс курсор привязан:
93
94
95
96      ПерБлБит копировать биты
        ПерБлБит снова копировать биты
97      Словарь системы элеметарный метод снимок
98      Время класс
99      Время класс
100     Планировшик исполнителя
101     Курсор
102     Показываемый экран
103*    Сканер знаков
104*    ПерБлБит
105*    Ряд байтов
        Ряд байтов
        Цепь
        Цепь
106
```

```
107
108
109
110     Знак
        Объект
111     Объект
112     Словарь системы
113     Словарь системы
114     Словарь системы
115     Словарь системы
116     Словарь системы
117
118
119
120
121
122
123
124
125
126
127
```

Пример элементарного вызова метода это ответ экземпляра *Малого целого* на сообщение с селектором +. Если аргумент также является экземпляром *Малого целого*, и сумма значений получателя и аргумента входит в диапазон представимый *Малом целым*, то элементарный метод удалит со стэка получателя и аргумент и заменит их на экземпляр *Малого целого* равный сумме. Если аргумент не является *Малым целым* или сумма выходит за представимый диапазон, то элементарный метод терпит неудачу и выполняется метод Смолтока связанный с селектором *Малого целого* +.

Используемые в этой книге определения управляющих структур и управляющие структуры используемые реализацией интерпретатора на машинном языке будут возможно использовать различные механизмы при неудаче элементарного метода. При возникновении условия неудачи элементарный метод на машинном языке может не

возвращать управление а просто прыгнуть в соответствующее место интерпретатора (обычно в место которое активирует *Откомпилированный метод*). Однако, формальное определение написана на Смолтоке, все процедуры должны вернуть управление своим отправителям и интерпретатор должен отслеживать удачу или неудачу элементарной процедуры независимо от структуры вызова процедур. Поэтому в определение включён регистр *успех* инициализируемый истиной при начале выполнения элементарной процедуры и ему может быть присвоена ложь при неудаче. Следующие две процедуры устанавливают и проверяют состояние регистра успех.

```
успех: значение успеха
    успех ← значение успеха & успех.

успех
    ↑ успех.
```

Следующие процедуры устанавливают флаг успеха в двух наиболее частых случаях: инициализация до выполнения элементарной процедуры и элементарный метод определил что он не может завершиться успешно.

```
ини элементарный
    успех ← истина.

неудача элементарного метода
    успех ← ложь.
```

Большинство элементарных процедур манипулируют целыми значениями, поэтому интерпретатор содержит несколько процедур выполняющих общие функции. Процедура *вытолкнуть целое* используется когда элементарная процедура ожидает на вершине стэка *Малое целое*. Если вершина это *Малое целое*, то возвращается его значение, если нет, то сообщается о неудаче элементарной процедуры.

```
вытолкнуть целое
    | указатель целого |
    указатель целого ← сам вытолкнуть стэк.
    сам успех: ( память это объект целое: указатель целого ).
    сам успех истина: [ ↑ память значение целого для: указатель цело-
```

го.].

Заметьте что процедура *достать целое:из объекта:* сигнализирует о неудаче если указанное поле не содержит *Малого целого*. Процедура *протолкнуть целое:* преобразовывает значение в *Малое целое* и помещает его на стэк.

```
протолкнуть целое: значение целого
  сам протолкнуть: (память объект целое для: значение целого).
```

Т.к. наибольший нумерованный набор может содержать 65534 элементов, и *Малые целые* могут представлять значения до 16383, то элементарные процедуры работающие с номерами или размерами должны уметь манипулировать *Большими положительными целыми*. Следующие две процедуры преобразуют в обе стороны между 16-ти битными беззнаковыми значениями и указателями на *Малые целые* или на *Большие положительные целые*.

```
положительное 16 битное целое для: значение целого
  | новое большое целое |
 значение целого < 0 истина: [↑сам неудача элементарного метода.].
  (память это значение целого: значение целого)
    истина: [↑память объект целое для: значение целого.].
  новое большое целое ← память
     экземпляр класса: Указатель на класс Большое положительное целое
     с байтами: 2.
  память
     сохранить байт: 0
     в объект: новое большое целое
     со значением: (сам младший байт: значение целого).
  память
     сохранить байт: 1
     в объект: новое большое целое
     со значением: (сам старший байт: значение целого).
  ↑новое большое целое.

положительное 16 битное значение для: указатель целого
  | значение |
  (память это объект целое: указатель целого)
```

```
        истина: [↑память значение целого для: указатель целого.].
    (память извлечь класс: указатель целого)
        = Указатель на класс Большое положительное целое
            ложь: [↑сам неудача элементарного метода.].
    (память извлечь длину в байтах: указатель целого) = 2
        ложь: [↑сам неудача элементарного метода.].
    значение ← память извлечь байт: 1 из объекта: указатель целого.
    значение ← значение * 256 + (память извлечь байт: 0 из объекта:
указатель целого).
    ↑значение.
```

Есть три способа которыми в процессе интерпретации байткодов посылки сообщений могут быть вызваны элементарные процедуры.

1. Некоторые элементарные процедуры связаны с байткодами посылки специального селектора для определённых классов получателей. Они могут быть вызваны без поиска сообщения.

2. Две наиболее частых элементарных процедуры (возвращение *себя* или переменной экземпляра) могут быть указаны в значении флага заголовка *Откомпилированного метода*. Они находятся только после того как поиск сообщения возвращает *Откомпилированный метод*, но для этого нужно только проверить заголовок.

3. Большинство элементарных процедур указываются номером в расширении заголовка *Откомпилированного метода*. Они находятся после поиска сообщения.

Первый способ обращения к элементарной процедуре представляется вызовом процедуры *ответ элементарного метода специального селектора* из процедуры *байткод послать специальный селектор*. Процедура *ответ элементарного метода специального селектора* выбирает подходящую элементарную процедуру и возвращает *истину* если успешно был выполнен элементарный ответ или *ложь* иначе. Заметьте что процедура *байткод послать специальный селектор* ищет специальный селектор если процедура *ответ элементарного метода специального селектора* вернула *ложь*.

ответ элементарного метода специального селектора

```
сам ини элементарный.
(текущий байткод между: 176 и: 191)
  истина: [сам элементарный метод арифметический селектор.].
(текущий байткод между: 192 и: 207)
  истина: [сам элементарный метод общий селектор.].
↑сам успех.
```

Элементарная процедура может быть вызвана специальным арифметическим селектором только если получатель это *Малое целое*. Реальные элементарные процедуры будут описаны в разделе об арифметических элементарных метода.

элементарный метод арифметический селектор

```
сам
  успех: (память это объект целое: (сам значение стэка: 1)).
сам успех
  истина: [
    текущий байткод = 176
      истина: [↑сам элементарный добавить.].
    текущий байткод = 177
      истина: [↑сам элементарный вычесть.].
    текущий байткод = 178
      истина: [↑сам элементарный меньше чем.].
    текущий байткод = 179
      истина: [↑сам элементарный больше чем.].
    текущий байткод = 180
      истина: [↑сам элементарный меньше или равно.].
    текущий байткод = 181
      истина: [↑сам элементарный больше или равно.].
    текущий байткод = 182
      истина: [↑сам элементарный равно.].
    текущий байткод = 183
      истина: [↑сам элементарный не равно.].
    текущий байткод = 184
      истина: [↑сам элементарный умножить.].
    текущий байткод = 185
      истина: [↑сам элементарный разделить.].
```

```
        текущий байткод = 186
            истина: [↑сам элементарный деление по модулю.].
        текущий байткод = 187
            истина: [↑сам элементарный создать точку.].
        текущий байткод = 188
            истина: [↑сам элементарный сдвинуть биты.].
        текущий байткод = 189
            истина: [↑сам элементарный частное.].
        текущий байткод = 190
            истина: [↑сам элементарный побитовое и.].
        текущий байткод = 191
            истина: [↑сам элементарный побитовое или.].].
```

Только пять из не арифметических специальных селекторов вызывают элементарную процедуру без поиска сообщения (==, *класс*, *экземпляр блока:*, *значение* и *значение:*). Элементарная процедура для == находится в разделе о элементарных сообщениях системы а процедуры для *класса* в элементарных методах управления памятью. Обе эти процедуры вызываются для любого класса получателя. Процедуры для *экземпляра блока:*, *значения* и *значения:* находятся в разделе управляющих элементарных методов. Процедура для *экземпляра блока:* вызывается если получатель это *Контекст метода* или *Контекст блока*. Процедуры для *значения* и *значения:* вызываются только если получатель это *Контекст блока*.

элементарный метод общий селектор

```
    | класс получателя |
    количество аргументов ← сам
        достать целое: текущий байткод − 176 * 2 + 1
        из объекта: Указатель на специальные селекторы.
    класс получателя ← память
        извлечь класс: ( сам значение стэка: количество аргументов ).
    текущий байткод = 198
        истина: [↑сам элементарный метод эквивалентность.].
    текущий байткод = 199
        истина: [↑сам элементарный метод класс.].
    текущий байткод = 200
        истина: [
```

```
        сам
          успех: класс получателя
            = Указатель на класс Контекст метода
              | ( класс получателя
                  = Указатель на класс Контекст блока ).
        ↑сам успех
          истина: [сам элментарный метод экземпляр блока. ]. ].
    текущий байткод = 201 | ( текущий байткод = 202 )
      истина: [
        сам
          успех: класс получателя
            = Указатель на класс Контекст блока.
        ↑сам успех
          истина: [сам значение элементарного метода. ]. ].
    сам неудача элементарного метода.
```

Вторая и третья часть элементарный процедур указынных выше вызываются после получения для сообщения *Откомпилированного метода*. Наличие элементарного метода определяется процедурой *ответ элементарного метода* вызываемой процедурой *выполнить новый метод*. Процедура *ответ элементарного метода* подобна процедуре *ответ элементарного метода специального селектора* которая возвращает *истину* если элементарная процедура завершилась успешно и возвращает *ложь* иначе. Заметьте что процедура *выполнить новый метод* активирует *Откомпилированный метод* который ищется если *ответ элементарного метода* возвращает *ложь*.

ответ элементарного метода

```
  | значение флага этот получатель смещение |
  номер элементарного метода = 0
    истина: [
      значение флага ← сам значение флага: новый метод.
      значение флага = 5
        истина: [сам быстрый возврат себя. ↑истина. ].
      значение флага = 6
        истина: [
          сам быстрая загрузка переменной экземпляра.
```

```
            ↑истина. ].
        ↑ложь. ]
    ложь: [
        сам ини элементарный.
        сам выполнить элементарный метод.
        ↑сам успех. ].
```

Значения флага 5 и 6 соответствуют двум наиболее часто используемым элементарным процедурам, простому возвращению *себя* и простому возвращению переменной экземпляра. Возвращение *себя* это отсутствие операции т.к. интерпретатор гарантирует что указатель на объект *сам* занимает то же место на стэке которое должен занимать получатель сообщения в качестве ответа на сообщение.

быстрый возврат себя

Сложность возвращение переменной экземпляра получателя не очень большая.

быстрая загрузка переменной экземпляра

```
| этот получатель номер поля |
этот получатель ← сам вытолкнуть стэк.
номер поля ← сам номер поля для: новый метод.
сам
    протолкнуть: (память
        извлечь указатель: номер поля
        из объекта: этот получатель ).
```

Шесть типов элементарных процедур формального определения работают с арифметикой, с индексами и потоками, с управлением памятью, с управляющими структурами, с вводом/выводом и с общим доступом к системе. Эти типы соответствуют шести интервалам номеров элементарных процедур. Интервал номеров элементарных процедур был зарезервирован для реализации собственных элементарных процедур. Он может иметь любой смысл, но может зависеть от интерпретатора. Т.к. он не является частью определения, то он не рассматривается здесь.

выполнить элементарный метод

```
номер элементарного метода < 60
```

```
        истина: [↑сам выполнить арифметический элементарный ме-
тод.].
    номер элементарного метода < 68
        истина: [
            ↑сам
                выполнить элементарные метоы нумерации и Потоков.].
    номер элементарного метода < 80
        истина: [
            ↑сам
                выполнить элементарный метод управления хранилищем.].
    номер элементарного метода < 90
        истина: [↑сам выполнить управляющий элементарный метод.].
    номер элементарного метода < 110
        истина: [↑сам выполнить элементарный метод ввода вывода.].
    номер элементарного метода < 128
        истина: [↑сам выполнить элементарный метод системы.].
    номер элементарного метода < 256
        истина: [↑сам выполнить собственный элементарный метод.].
```

29.1 Арифметические элементарные методы

Есть три набора арифметических элементарных процедур, один для *Малых целых*, один для болших целых (*Большое положительное целое* и *Большое отрицательное целое*) и один для *Плавающих*. Элементарные методы для *Малых целых* и *Плавающих* должны быть реализованы, элементарные методы для больших целых необязательны.

выполнить арифметический элементарный метод

```
    номер элементарного метода < 20
        истина: [↑сам выполнить элементарный метод целого.].
    номер элементарного метода < 40
        истина: [
            ↑сам выполнить элементарный метод большого целого.].
    номер элементарного метода < 60
        истина: [↑сам выполнить элементарный метод плавающего.].
```

Все арифметические элементарные процедуры из первого набора снимают со стэка получателя и аргумент и заканчиваются неудачей если два эти значения не являются *Малыми целыми*. Затем процедуры помещают на стэк или цело результат вычислений или *Логический* результат сравнения. Процедуры производящие целое завершаются неудачно если значение результата не представимо *Малым целым*.

```
выполнить элементарный метод целого
    номер элементарного метода = 1
        истина: [↑сам элементарный добавить.].
    номер элементарного метода = 2
        истина: [↑сам элементарный вычесть.].
    номер элементарного метода = 3
        истина: [↑сам элементарный меньше чем.].
    номер элементарного метода = 4
        истина: [↑сам элементарный больше чем.].
    номер элементарного метода = 5
        истина: [↑сам элементарный меньше или равно.].
    номер элементарного метода = 6
        истина: [↑сам элементарный больше или равно.].
    номер элементарного метода = 7
        истина: [↑сам элементарный равно.].
    номер элементарного метода = 8
        истина: [↑сам элементарный не равно.].
    номер элементарного метода = 9
        истина: [↑сам элементарный умножить.].
    номер элементарного метода = 10
        истина: [↑сам элементарный разделить.].
    номер элементарного метода = 11
        истина: [↑сам элементарный деление по модулю.].
    номер элементарного метода = 12
        истина: [↑сам элементарный разд.].
    номер элементарного метода = 13
        истина: [↑сам элементарный частное.].
    номер элементарного метода = 14
        истина: [↑сам элементарный побитовое и.].
    номер элементарного метода = 15
```

```
    истина: [↑сам элементарный побитовое или.].
номер элементарного метода = 16
    истина: [↑сам элементарный побитовое ислючающее или.].
номер элементарного метода = 17
    истина: [↑сам элементарный сдвинуть биты.].
номер элементарного метода = 18
    истина: [↑сам элементарный создать точку.].
```

Процедуры *элементарный добавить*, *элементарный вычесть* и *элементарный умножить* идентичны за исключением используемой операции, поэтому здесь будет показана только процедура *элементарный добавить*.

элементарный добавить

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех
    истина: [
        результат целое ← получатель целое + аргумент целое.
        сам
            успех: (память это значение целого: результат целое).].
сам успех
    истина: [сам протолкнуть целое: результат целое.]
    ложь: [сам отменить выталкивание: 2.].
```

Элементарная процедура для деления (связанная с селектором /) отличается от других трёх арифметических элементарных процедур т.к. она возвращает результат только если он точный, иначе происходит неудача. Эта элементарная процедура и следующие три, работающие с делением с округлением, завершаются неудачно если аргумент равен нулю.

элементарный разделить

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех: аргумент целое ~= 0.
сам успех: получатель целое \\ аргумент целое = 0.
```

```
сам успех
  истина: [
    результат целое ← получатель целое // аргумент целое.
    сам
      успех: ( память это значение целого: результат целое ). ].
сам успех
  истина: [
    сам
      протолкнуть: ( память объект целое для: результат целое ). ]
  ложь: [сам отменить выталкивание: 2. ].
```

Элементарная процедура для деления по модулю (связанная с селектором \\) возвращает остаток деления с частным округлённым вниз (к минус бесконечности).

элементарный деление по модулю

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех: аргумент целое ~= 0.
сам успех
  истина: [
    результат целое ← получатель целое \\ аргумент целое.
    сам
      успех: ( память это значение целого: результат целое ). ].
сам успех
  истина: [сам протолкнуть целое: результат целое. ]
  ложь: [сам отменить выталкивание: 2. ].
```

Есть две элементарных процедуры для округлённого деления (связанные с селекторами // и *частное:*). Результат // всегда округлён книзу (к минус бесконечности).

элементарный разд

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех: аргумент целое ~= 0.
сам успех
```

```
    истина: [
      результат целое ← получатель целое // аргумент целое.
      сам
        успех: ( память это значение целого: результат целое ). ].
  сам успех
    истина: [сам протолкнуть целое: результат целое. ]
    ложь: [сам отменить выталкивание: 2. ].
```

Результат *частного:* урезан (округлён к нулю).

элементарный частное

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех: аргумент целое ~= 0.
сам успех
  истина: [
    результат целое ← получатель целое частное: аргумент целое.
    сам
      успех: ( память это значение целого: результат целое ). ].
сам успех
  истина: [сам протолкнуть целое: результат целое. ]
  ложь: [сам отменить выталкивание: 2. ].
```

Все процедуры *элементарный равно*, *элементарный не равно*, *элементарный меньше чем*, *элементарный меньше или равно*, *элементарный больше чем* и *элементарный больше или равно* идентичны за исключением используемой операции сравнения, поэтому ниже показана только процедура *элементарный равно*.

элементарный равно

```
| получатель целое аргумент целое результат целое |
аргумент целое ← сам вытолкнуть целое.
получатель целое ← сам вытолкнуть целое.
сам успех
  истина: [
    получатель целое = аргумент целое
      истина: [сам протолкнуть: Указатель на истину. ]
      ложь: [сам протолкнуть: Указатель на ложь. ]. ]
```

```
    ложь: [сам отменить выталкивание: 2.].
```

Процедуры *элементарный побитовое и*, *элементарный побитовое или* и *элементарный побитовое исключающее или* выполняют логические операции над значениями *Малых целых* в двоичном представлении дополнения до двух. Они идентичны за исключением используемой логической операции , поэтому ниже показана только процедура *элементарный побитовое и*.

элементарный побитовое и

```
| получатель целое аргумент целое результат целое |
аргумент целое ⟵ сам вытолкнуть целое.
получатель целое ⟵ сам вытолкнуть целое.
сам успех
  истина: [
   результат целое ⟵ получатель целое побитовое и: аргумент целое.].
сам успех
  истина: [сам протолкнуть целое: результат целое.]
  ложь: [сам отменить выталкивание: 2.].
```

Элементарная процедура сдвига (связанная с селектором *сдвинуть биты:*) возвращает *Малое целое* в двоичном коде дополнения до двух представляющее получателя сдвинутого влево на величину указанную аргументом. Отрицательный аргумент сдвигает вправо. При сдвиге влево справа добавляются нули. При сдвиге вправо биты слева заполняются знаковым битом. Этот элементарный метод заканчивается неудачно если результат нельзя представить *Малым целым*.

элементарный сдвинуть биты

```
| получатель целое аргумент целое результат целое |
аргумент целое ⟵ сам вытолкнуть целое.
получатель целое ⟵ сам вытолкнуть целое.
сам успех
  истина: [
      результат целое ⟵ получатель целое сдвинуть биты: аргумент целое.
      сам
```

```
        успех: (память это значение целого: результат целое).].
  сам успех
    истина: [сам протолкнуть целое: результат целое.]
    ложь: [сам отменить выталкивание: 2.].
```

Элементарная процедура связанная с селектором @ возвращает новую *Точку* чьё значение икс это получатель а значение игрек это аргумент.

элементарный создать точку

```
  | получатель целое аргумент целое результат точка |
  аргумент целое ⟵ сам вытолкнуть стэк.
  получатель целое ⟵ сам вытолкнуть стэк.
  сам
    успех: (память это значение целого: получатель целое).
  сам успех: (память это значение целого: аргумент целое).
  сам успех
    истина: [
      результат точка ⟵ память
        экземпляр класса: Указатель на класс Точка
        с указателями: Размер класса Точка.
      память
        сохранить указатель: Номер икса
        в объект: результат точка
        со значением: получатель целое.
      память
        сохранить указатель: Номер игрека
        в объект: результат точка
        со значением: аргумент целое.
      сам протолкнуть: результат точка.]
    ложь: [сам отменить выталкивание: 2.].
```

инициализировать номера Точки

```
  Номер икса ⟵ 0.
  Номер игрека ⟵ 1.
  Размер класса Точка ⟵ 2.
```

Элементарные процедуры с номерами от 21 до 37 те же что и для номеров от 1 до 17 за исключением того что они выполняют

операции на большими целыми (экземплярами *Большого положительного целого* и *Большого отрицательного целого*). Есть соответствующие реализации на Смолтоке для всех этих операций, поэтому элементарные процедуры не обязательны и не будут определяться в этой главе. Чтобы реализовать их нужно перевести на машинный язык соответствующие методы Смолтока.

выполнить элементарный метод большого целого
сам неудача элементарного метода.

Экземпляры *Плавающего* представляются в формате IEEE с одинарной точностью (32 бита). Формат представляет величину с плавающей точкой как число между еденицей и двойкой, степени двойки и знака. *Плавающее* это объект неуказатель размером в слово. Наиболее значимый бит первого поля указывает знак числа (1 — знак минус). Следующие восемь наиболее значимых битов первого поля это 8-ми битная экспонента смещённая на 127 (0 означает значение экспоненты -127, 128 означает экспоненту 1 и т.д.). Семь младших битов первого поля это семь наиболее значимых битов дробной части числа между единицей и двойкой. Длина дробной части 23 бита и 16 младших битов это второе поле *Плавающего*. Поэтому *Плавающее* чьи поля это:

ЗЭЭЭЭЭЭЭ ЭПППППП
ППППППП ППППППП

представляет значение
$-1^З * 2^{Э-127} * 1.П$
0 представляется обоими полями равными нулю. Элементарные методы заканчиваются неудачно если аргумент не является экземпляром *Плавающего* или если результат не представим как *Плавающее*. Это определение виртуальной машины Смолтока не содержит частей стандарта IEEE кроме представления плавающих числе. Реализация процедур выполняющих требуемые операции над значениями с плавающей запятой оставлена реализующим систему.

Элементарная процедура *элементарный как плавающее* преобразовывает получателя в *Плавающее*. Процедуры с номерами от 41 до 50 выполняют те же самые операции что и процедуры от 1 до

10 и от 21 до 30, за исключением того что они работают с *Плавающими*. Процедура *элементарный плавающее усечь* возвращает *Малое целое* равное значению получателя без дробной части. Она заканчивается неудачно если значение нельзя представить *Малым целым*. Процедура *элементарный дробная часть* возвращает разность между получателем и его усечённым значением. Процедура *элементарный экспонента* возвращает экспоненту получателя, а процедура *элементарный умножить на два в степени* увеличивает экспоненту на количество заданное аргументом.

```
выполнить элементарный метод плавающего
    номер элементарного метода = 40
        истина: [↑сам элементарный как плавающее.].
    номер элементарного метода = 41
        истина: [↑сам элементарный добавить плавающее.].
    номер элементарного метода = 42
        истина: [↑сам элементарный вычесть плавающее.].
    номер элементарного метода = 43
        истина: [↑сам элементарный плавающее меньше чем.].
    номер элементарного метода = 44
        истина: [↑сам элементарный плавающее больше чем.].
    номер элементарного метода = 45
        истина: [↑сам элементарный плавающее меньше или равно.].
    номер элементарного метода = 46
        истина: [↑сам элементарный плавающее больше или равно.].
    номер элементарного метода = 47
        истина: [↑сам элементарный плавающее равно.].
    номер элементарного метода = 48
        истина: [↑сам элементарный плавающее не равно.].
    номер элементарного метода = 49
        истина: [↑сам элементарный плавающее умножить.].
    номер элементарного метода = 50
        истина: [↑сам элементарный плавающее разделить.].
    номер элементарного метода = 51
        истина: [↑сам элементарный плавающее усечь.].
    номер элементарного метода = 52
        истина: [↑сам элементарный дробная часть.].
    номер элементарного метода = 53
```

```
    истина: [↑сам элементарный экспонента.].
  номер элементарного метода = 54
    истина: [↑сам элементарный умножить на два в степени.].
```

29.2 Элементарные методы *Ряда* и *Потока*

Второй набор элементарных процедур предназначен для работы с нумерованными полями объектов как напрямую, при помощи номеров, так и косвенно, через потоки. Эти процедуры используют процедуры 16-ти битных положительных целых, т.к. предел нумерованных полей равен 65534.

выполнить элементарные метоы нумерации и Потоков

```
  номер элементарного метода = 60
    истина: [↑сам элементарный от.].
  номер элементарного метода = 61
    истина: [↑сам элементарный от пом.].
  номер элементарного метода = 62
    истина: [↑сам элементарный размер.].
  номер элементарного метода = 63
    истина: [↑сам элементарный Цепь от.].
  номер элементарного метода = 64
    истина: [↑сам элементарный Цепь от пом.].
  номер элементарного метода = 65
    истина: [↑сам элементарный следующий.].
  номер элементарного метода = 66
    истина: [↑сам элементарный пом следующим.].
  номер элементарного метода = 67
    истина: [↑сам элементарный в конце.].
```

Следующие процедуры используются для проверки границ при нумеровании и для выполнения доступа к нумерованным полям. Они определяют содержит ли нумеруемый объект в своих полях 16-ти битные целые значения или 8-ми битные целые значения. Процедура *проверить границы нумерования для:в:* получает номер относительно единицы и определяет является ли он допустимым для

объекта. Она должна работать с любыми нумерованными полями.

```
проверить границы нумерования для: номер в: ряд
  | класс |
  класс ← память извлечь класс: ряд.
  сам успех: номер >= 1.
  сам
    успех: номер + ( сам фиксированных полей: класс )
      <= ( сам длина для: ряд ).

длина для: ряд
  ( сам это слова: ( память извлечь класс: ряд ) )
    истина: [ ↑ память извлечь длину в словах: ряд. ]
    ложь: [ ↑ память извлечь длину в байтах: ряд. ].
```

Процедуры *подномер:с:* *подномер:с:сохранить:* предполагают что к номеру было добавлено количество фиксированных полей, поэтому они используют его как номер относительно единицы для всего объекта.

```
подномер: ряд с: номер
  | класс значение |
  класс ← память извлечь класс: ряд.
  ( сам это слова: класс )
    истина: [
      ( сам это указатели: класс )
        истина: [
          ↑ память
            извлечь указатель: номер - 1
            из объекта: ряд. ]
        ложь: [
          значение ← память
            извлечь слово: номер - 1
            из объекта: ряд.
          ↑ сам
            положительное 16 битное целое для: значение. ]. ]
    ложь: [
      значение ← память извлечь байт: номер - 1 из объекта: ряд.
      ↑ память объект целое для: значение. ].
```

```
подномер: ряд с: номер сохранить: значение
  | класс |
  класс ← память извлечь класс: ряд.
  ( сам это слова: класс )
    истина: [
      ( сам это указатели: класс )
        истина: [
          ↑ память
            сохранить указатель: номер — 1
            в объект: ряд
            со значением: значение. ]
        ложь: [
          сам
            успех: ( память это объект целое: значение ).
          сам успех
            истина: [
              ↑ память
                сохранить слово: номер — 1
                в объект: ряд
                со значением: ( сам
                        положительное 16 битное значение для:
значение ). ]. ]. ]
    ложь: [
      сам успех: ( память это объект целое: значение ).
      сам успех
        истина: [
          ↑ память
            сохранить байт: номер — 1
            в объект: ряд
            со значением: ( сам
                    младший байт: ( память значение целого для:
значение ) ). ]. ].
```

Процедуры *элементарный от* и *элементарный от пом* просто извлекают и помещают одно из нумерованных полей получателя. Они заканчиваются неудачно если номер это не *Малое целое* или если он выходит за границы.

элементарный от
```
| номер ряд класс ряда результат |
номер ← сам
  положительное 16 битное значение для: сам вытолкнуть стэк.
ряд ← сам вытолкнуть стэк.
класс ряда ← память извлечь класс: ряд.
сам проверить границы нумерования для: номер в: ряд.
сам успех
  истина: [
    номер ← номер + ( сам фиксированных полей: класс ряда ).
    результат ← сам подномер: ряд с: номер. ].
сам успех
  истина: [сам протолкнуть: результат. ]
  ложь: [сам отменить выталкивание: 2. ].
```

Процедура *элементарный от пом* также заканчивается неудачно если получатель тип получателя не указатель и второй аргумент это не 8-ми битное положительное целое (для объектов нумеруемых побайтно) или 16-ти битное положительное целое (для объектов нумеруемых пословно). Процедура возвращает помещаемое значение.

элементарный от пом
```
| ряд номер класс ряда значение результат |
значение ← сам вытолкнуть стэк.
номер ← сам
  положительное 16 битное значение для: сам вытолкнуть стэк.
ряд ← сам вытолкнуть стэк.
класс ряда ← память извлечь класс: ряд.
сам проверить границы нумерования для: номер в: ряд.
сам успех
  истина: [
    номер ← номер + ( сам фиксированных полей: класс ряда ).
    сам подномер: ряд с: номер сохранить: значение. ].
сам успех
  истина: [сам протолкнуть: значение. ]
  ложь: [сам отменить выталкивание: 3. ].
```

Процедура *элементарный размер* возвращает количество нумерованных полей получателя (т.е. наибольший позволимый номер

поля).

элементарный размер

```
| ряд класс длина |
ряд ← сам вытолкнуть стэк.
класс ← память извлечь класс: ряд.
длина ← сам
  положительное 16 битное целое для: ( сам длина для: ряд )
    – ( сам фиксированных полей: класс ).
сам успех
  истина: [ сам протолкнуть: длина. ]
  ложь: [ сам отменить выталкивание: 1. ].
```

Процедуры *элементарный Цепь от* и *элементарный Цепь от пом* являются специальными ответами на сообщения *от:* и *от:пом:* посланные экземплярам *Цепи*. В действительности *Цепи* хранят 8-ми битные числа в полях нумеруемых побайтно, но они возвращают *Знаки* в ответ на сообщения *от:* и *от:пом:*. У *Знака* есть одна переменная экземпляра которая хранит *Малое целое*. Значение *Малого целого* возвращённого сообщением *от:* это байт хранящийся в указанном поле *Цепи*. Процедура *элементарный Цепь от* всегда возвращает один и тот же экземпляр *Знака* для данного значения. Она берёт *Знаки* из *Ряда* в памяти объектов который является гарантированным указателем объекта называемым *Указатель на таблицу знаков*.

элементарный Цепь от

```
| номер ряд сакои знак |
номер ← сам
  положительное 16 битное значение для: сам вытолкнуть стэк.
ряд ← сам вытолкнуть стэк.
сам проверить границы нумерования для: номер в: ряд.
сам успех
  истина: [
    сакои ← память
      значение целого для: ( сам подномер: ряд с: номер ).
    знак ← память
      извлечь указатель: сакои
      из объекта: Указатель на таблицу знаков. ].
```

```
  сам успех
    истина: [сам протолкнуть: знак.]
    ложь: [сам отменить выталкивание: 2.].
```

инициализировать номера Знака

```
  Номер значения Знака ← 0.
```

Процедура *элементарный Цепь от пом* сохраняет значение *Знака* в один из нумерованных байтов получателя. Она заканчивается неудачно если второй аргумент сообщения *от:пом:* это не *Знак*.

элементарный Цепь от пом

```
  | номер ряд сакои знак |
  знак ← сам вытолкнуть стэк.
  номер ← сам
    положительное 16 битное значение для: сам вытолкнуть стэк.
  ряд ← сам вытолкнуть стэк.
  сам проверить границы нумерования для: номер в: ряд.
  сам
    успех: (память извлечь класс: знак) = Указатель на класс Знак.
  сам успех
    истина: [
      сакои ← память
        извлечь указатель: Номер значения Знака
        из объекта: знак.
      сам подномер: ряд с: номер сохранить: сакои.].
  сам успех
    истина: [сам протолкнуть: знак.]
    ложь: [сам отменить выталкивание: 2.].
```

Процедуры *элементарный следующий*, *элементарный пом следующим* и *элементарный в конце* являются необязательными элементарными для методов потоков *следующий*, *пом следующим:* и *в конце*. Процедуры *элементарный следующий* и *элементарный пом следующим* работают только если объект потока это *Ряд* или *Цепь*.

инициализировать номера Потока

```
  Номер ряда потока ← 0.
```

```
Номер номера Потока ← 1.
Номер предела чтения Потока ← 2.
Номер предела записи Потока ← 3.

элементарный следующий
| поток номер предел ряд класс ряда результат сакои |
поток ← сам вытолкнуть стэк.
ряд ← память
  извлечь указатель: Номер ряда потока
  из объекта: поток.
класс ряда ← память извлечь класс: ряд.
номер ← сам достать целое: Номер номера Потока из объекта: поток.
предел ← сам
  достать целое: Номер предела чтения Потока
  из объекта: поток.
сам успех: номер < предел.
сам
  успех: класс ряда = Указатель на класс Ряд
    | (класс ряда = Указатель на класс Цепь).
сам проверить границы нумерования для: номер + 1 в: ряд.
сам успех
  истина: [
    номер ← номер + 1.
    результат ← сам подномер: ряд с: номер.].
сам успех
  истина: [
    сам
      сохранить целое: Номер номера Потока
      в объект: поток
      со значением: номер.].
сам успех
  истина: [
    класс ряда = Указатель на класс Ряд
      истина: [сам протолкнуть: результат.]
      ложь: [
        сакои ← память значение целого для: результат.
```

```
            сам
              протолкнуть: (память
                извлечь указатель: сакои
                из объекта: Указатель на таблицу знаков).].]
      ложь: [сам отменить выталкивание: 1.].

элементарный пом следующим
  | значение поток номер предел ряд класс ряда результат сакои |
  значение ← сам вытолкнуть стэк.
  поток ← сам вытолкнуть стэк.
  ряд ← память
    извлечь указатель: Номер ряда потока
    из объекта: поток.
  класс ряда ← память извлечь класс: ряд.
  номер ← сам достать целое: Номер номера Потока из объекта: по-
ток.
  предел ← сам
    достать целое: Номер предела записи Потока
    из объекта: поток.
  сам успех: номер < предел.
  сам
    успех: класс ряда = Указатель на класс Ряд
      | (класс ряда = Указатель на класс Цепь).
  сам проверить границы нумерования для: номер + 1 в: ряд.
  сам успех
    истина: [
      номер ← номер + 1.
      класс ряда = Указатель на класс Ряд
        истина: [
          сам
            подномер: ряд
            с: номер
            сохранить: значение.]
        ложь: [
          сакои ← память
            извлечь указатель: Номер значения Знака
            из объекта: значение.
```

```
            сам подномер: ряд с: номер сохранить: сакои. ]. ].
  сам успех
    истина: [
      сам
        сохранить целое: Номер номера Потока
        в объект: поток
        со значением: номер. ].
  сам успех
    истина: [сам протолкнуть: значение. ]
    ложь: [сам отменить выталкивание: 2. ].

элементарный в конце
  | поток ряд класс ряда длина номер предел |
  поток ← сам вытолкнуть стэк.
  ряд ← память
    извлечь указатель: Номер ряда потока
    из объекта: поток.
  класс ряда ← память извлечь класс: ряд.
  длина ← сам длина для: ряд.
  номер ← сам достать целое: Номер номера Потока из объекта: по-
ток.
  предел ← сам
    достать целое: Номер предела чтения Потока
    из объекта: поток.
  сам
    успех: класс ряда = Указатель на класс Ряд
      | ( класс ряда = Указатель на класс Цепь ).
```

29.3 Элементарные методы управления памятью

The storage management primitive routines manipulate the representation of objects. They include primitives for manipulating object pointers, accessing fields, creating new instances of a class, and enumerating the instances of a class.

выполнить элементарный метод управления хранилищем

```
номер элементарного метода = 68
    истина: [↑сам элементарный Объект от.].
номер элементарного метода = 69
    истина: [↑сам элементарный Объект от пом.].
номер элементарного метода = 70
    истина: [↑сам элементарный новый.].
номер элементарного метода = 71
    истина: [↑сам элементарный новый с аргументом.].
номер элементарного метода = 72
    истина: [↑сам элементарный становится.].
номер элементарного метода = 73
    истина: [↑сам элементарный пер экз от.].
номер элементарного метода = 74
    истина: [↑сам элементарный пер экз от пом.].
номер элементарного метода = 75
    истина: [↑сам элементарный как УО.].
номер элементарного метода = 76
    истина: [↑сам элементарный как объект.].
номер элементарного метода = 77
    истина: [↑сам элементарный некоторый экземпляр.].
номер элементарного метода = 78
    истина: [↑сам элементарный следующий экземпляр.].
номер элементарного метода = 79
    истина: [↑сам элементарный новый метод.].
```

The primitiveObjectAt and primitiveObjectAtPut routines are associated with the objectAt: and objectAt:put: messages in CompiledMethod. They provide access to the object pointer fields of the receiver (the method header and the literals) from Smalltalk. The header is accessed with an index of 1 and the literals are accessed with indices 2 through the number of literals plus 1. These messages are used primarily by the compiler.

элементарный Объект от

```
| этот получатель номер |
номер ← сам вытолкнуть целое.
этот получатель ← сам вытолкнуть стэк.
сам успех: номер > 0.
```

```
    сам
        успех: номер
            <= ( сам
                количество указателей объектов для: этот получатель ).
    сам успех
        истина: [
            сам
                протолкнуть: ( память
                    извлечь указатель: номер - 1
                    из объекта: этот получатель ). ]
        ложь: [сам отменить выталкивание: 2.].
```

элементарный Объект от пом

```
    | этот получатель номер новое значение |
    новое значение ← сам вытолкнуть стэк.
    номер ← сам вытолкнуть целое.
    этот получатель ← сам вытолкнуть стэк.
    сам успех: номер > 0.
    сам
        успех: номер
            <= ( сам
                количество указателей объектов для: этот получатель ).
    сам успех
        истина: [
            память
                сохранить указатель: номер - 1
                в объект: этот получатель
                со значением: новое значение.
            сам протолкнуть: новое значение. ]
        ложь: [сам отменить выталкивание: 3.].
```

The primitiveNew routine creates a new instance of the receiver (a class) without indexable fields. The primitive fails if the class is indexable.

элементарный новый

```
    | класс размер |
    класс ← сам вытолкнуть стэк.
```

```
размер ← сам фиксированных полей: класс.
сам успех: (сам это нумерованный: класс) == ложь.
сам успех
  истина: [
    (сам это указатели: класс)
      истина: [
        сам
          протолкнуть: (память
            экземпляр класса: класс
            с указателями: размер).]
      ложь: [
        сам
          протолкнуть: (память
            экземпляр класса: класс
            со словами: размер).].]
  ложь: [сам отменить выталкивание: 1.].
```

The primitiveNewWithArg routine creates a new instance of the receiver (a class) with the number of indexable fields specified by the integer argument. The primitive fails if the class is not indexable.

элементарный новый с аргументом

```
| размер класс |
размер ← сам
  положительное 16 битное значение для: сам вытолкнуть стэк.
класс ← сам вытолкнуть стэк.
сам успех: (сам это нумерованный: класс).
сам успех
  истина: [
    размер ← размер + (сам фиксированных полей: класс).
    (сам это указатели: класс)
      истина: [
        сам
          протолкнуть: (память
            экземпляр класса: класс
            с указателями: размер).]
      ложь: [
        (сам это слова: класс)
```

```
                истина: [
                  сам
                    протолкнуть: ( память
                      экземпляр класса: класс
                      со словами: размер ). ]
                ложь: [
                  сам
                    протолкнуть: ( память
                      экземпляр класса: класс
                      с байтами: размер ). ]. ]
    ложь: [сам отменить выталкивание: 2. ].
```

The primitiveBecome routine swaps the instance pointers of the receiver and argument. This means that all objects that used to point to the receiver now point to the argument and vice versa.

элементарный становится

```
| этот получатель другой указатель |
другой указатель ← сам вытолкнуть стэк.
этот получатель ← сам вытолкнуть стэк.
сам успех: ( память это объект целое: другой указатель ) не.
сам успех: ( память это объект целое: этот получатель ) не.
сам успех
  истина: [
    память
      обменять указатель: этот получатель
      и: другой указатель.
    сам протолкнуть: этот получатель. ]
  ложь: [сам отменить выталкивание: 2. ].
```

The primitiveInstVarAt and primitiveInstVarAtPut routines are associated. with the instVarAt: and instVarAt:put: messages in Object. They are simi- |ar to primitiveAt and primitiveAtPut except that the numbering of fields starts with the fixed fields (corresponding to named instance variables) instead of with the indexable fields. The indexable fields are numbered starting with one more than the number of fixed fields. These routines need a different routine to check the bounds of the sUbscript.

проверить границы переменных экземпляра для: номер

```
в: объект
  | класс |
  класс ← память извлечь класс: объект.
  сам успех: номер >= 1.
  сам успех: номер <= ( сам длина для: объект ).

элементарный пер экз от
  | этот получатель номер значение |
  номер ← сам вытолкнуть целое.
  этот получатель ← сам вытолкнуть стэк.
  сам
    проверить границы переменных экземпляра для: номер
    в: этот получатель.
  сам успех
    истина: [значение ← сам подномер: этот получатель с: номер. ].
  сам успех
    истина: [сам протолкнуть: значение. ]
    ложь: [сам отменить выталкивание: 2. ].

элементарный пер экз от пом
  | этот получатель номер новое значение |
  новое значение ← сам вытолкнуть стэк.
  номер ← сам вытолкнуть целое.
  этот получатель ← сам вытолкнуть стэк.
  сам
    проверить границы переменных экземпляра для: номер
    в: этот получатель.
  сам успех
    истина: [
      сам
        подномер: этот получатель
        с: номер
        сохранить: новое значение. ].
  сам успех
    истина: [сам протолкнуть: новое значение. ]
    ложь: [сам отменить выталкивание: 3. ].
```

The primitiveAsOop routine produces a SmallInteger whose value is half of the receiver's object pointer (interpreting object pointers as 16-bit signed quantities). The primitive only works for non-SmallInteger receivers. Since non-SmallInteger object pointers are even, no information in the object pointer is lost. Because of the encoding of SmallIntegers, the halving operation can be performed by setting the least significant bit of the receiver's object pointer.

элементарный как УО

```
| этот получатель |
этот получатель ← сам вытолкнуть стэк.
сам
  успех: (память это объект целое: этот получатель) == ложь.
сам успех
  истина: [
    сам
      протолкнуть: (этот получатель побитовое или: 1).]
  ложь: [сам отменить выталкивание: 1.].
```

The primitiveAsObject routine performs the inverse operation of primitiveAsOop. It only works for SmallInteger receivers (it is associated with the asObject message in SmallInteger). It produces the object pointer that is twice the receiver's value. The primitive fails if there is no object for that pointer.

элементарный как объект

```
| этот получатель новый УО |
этот получатель ← сам вытолкнуть стэк.
новый УО ← этот получатель побитовое и: 16оЕЕЕД.
сам успех: (память содержит объект: новый УО).
сам успех
  истина: [сам протолкнуть: новый УО.]
  ложь: [сам отменить выталкивание: 1.].
```

The primitiveSomeInstance and primitiveNextInstance routines allow for the enumeration of the instances of a class. They rely on the ability of the object memory to define an ordering on object pointers, to find the first instance of a class in that ordering, and, given an object pointer, to find the next instance of the same class.

элементарный некоторый экземпляр

```
| класс |
класс ← сам вытолкнуть стэк.
(память экземпляры для: класс)
  истина: [
    сам
      протолкнуть: (память начальный экземпляр для: класс).]
  ложь: [сам неудача элементарного метода.].
```

элементарный следующий экземпляр

```
| объект |
объект ← сам вытолкнуть стэк.
(память это последний экземпляр: объект)
  истина: [сам неудача элементарного метода.]
  ложь: [
    сам протолкнуть: (память экземпляр после: объект).].
```

The primitiveNewMethod routine is associated with the newMethod:header: message in CompiledMethod class. Instances of CompitedMethod are created with a special message. Since the part of a CompiledMethod that contains pointers instead of bytes is indicated in the header, all CompiledMethods must have a valid header. Therefore, CompiledMethods are created with a message (newMethod:header:) that takes the number of bytes as the first argument and the header as the second argument. The header, in turn, indicates the number of pointer fields.

элементарный новый метод

```
| заголовок количество байткодов класс размер |
заголовок ← сам вытолкнуть стэк.
количество байткодов ← сам вытолкнуть целое.
класс ← сам вытолкнуть стэк.
размер ← (сам количество литералов заголовка: заголовок) + 1 * 2
  + количество байткодов.
сам
  протолкнуть: (память экземпляр класса: класс с байтами: размер).
```

29.4 Управляющие элементарные методы

The control primitives provide the control structures not provided by the bytecodes. They include support for the behavior of BlockConte?ts, Processes, and Semaphores. They also provide for messages with parameterized selectors.

выполнить управляющий элементарный метод

```
номер элементарного метода = 80
    истина: [↑сам элементарный экземпляр блока.].
номер элементарного метода = 81
    истина: [↑сам элементарный значение.].
номер элементарного метода = 82
    истина: [↑сам элементарный значение с аргументами.].
номер элементарного метода = 83
    истина: [↑сам элементарный выполнить.].
номер элементарного метода = 84
    истина: [↑сам элементарный выполнить с аргументами.].
номер элементарного метода = 85
    истина: [↑сам элементарный сигнал.].
номер элементарного метода = 86
    истина: [↑сам элементарный ждать.].
номер элементарного метода = 87
    истина: [↑сам элементарный возобновить.].
номер элементарного метода = 88
    истина: [↑сам элементарный приостановить.].
номер элементарного метода = 89
    истина: [↑сам элементарный очистить кэш.].
```

The primitiveBIockCopy routine is associated with the blockCopy: message in both BlockContext and MethodContext. This message is only produced by the compiler. The number of block arguments the new BlockConte?t takes is passed as the argument. The primitiveBIockCopy routine creates a new instance of BlockConte?t. If the receiver is a MethodContext, it becomes the new BlockContext's home context. If the receiver is a BlockConte?t, its home context is used for the new BlockConte?t's home context.

элементарный экземпляр блока

```
| контекст контекст метода количество аргументов блока новый контекст начальный УИ размер контекста |
  количество аргументов блока ← сам вытолкнуть стэк.
  контекст ← сам вытолкнуть стэк.
  ( сам это контекст блока: контекст )
    истина: [
      контекст метода ← память
        извлечь указатель: Номер дома
        из объекта: контекст. ]
    ложь: [ контекст метода ← контекст. ].
  размер контекста ← память извлечь длину в словах: контекст метода.
  новый контекст ← память
    экземпляр класса: Указатель на класс Контекст блока
    с указателями: размер контекста.
   начальный УИ ← память объект целое для: указатель инструкции + 3.
  память
    сохранить указатель: Номер начального УИ
    в объект: новый контекст
    со значением: начальный УИ.
  память
    сохранить указатель: Номер указателя инструкции
    в объект: новый контекст
    со значением: начальный УИ.
  сам
    сохранить значение указателя стэка: 0
    в контекст: новый контекст.
  память
    сохранить указатель: Номер количества аргументов блока
    в объект: новый контекст
    со значением: количество аргументов блока.
  память
    сохранить указатель: Номер дома
    в объект: новый контекст
    со значением: контекст метода.
  сам протолкнуть: новый контекст.
```

The primitiveValue routine is associated with all revalue"messages in BlockContext (value, value:, value:value:, and so on). It checks that the receiver takes the same number of block arguments that the "value"message did and then transfers them from the active context's stack to the receiver's stack. The primitive fails if the number of arguments do not match. The primitiveValue routine also stores the active context in the receiver's caller field and initializes the receiver's instruction pointer and stack pointer. After the receiver has been initialized, it becomes the active context.

```
элементарный значение
  | контекст блока количесво аргументов блока начальный УИ |
  контекст блока ← сам значение стэка: количество аргументов.
  количесво аргументов блока ← сам количество аргументов блока:
контекст блока.
  сам
    успех: количество аргументов = количесво аргументов блока.
  сам успех
    истина: [
      сам
        перенести: количество аргументов
        от номера: указатель стэка − количество аргументов + 1
        из объекта: активный контекст
        в номер: Начало блока временных
        в объект: контекст блока.
      сам вытолкнуть: количество аргументов + 1.
      начальный УИ ← память
        извлечь указатель: Номер начального УИ
        из объекта: контекст блока.
      память
        сохранить указатель: Номер указателя инструкции
        в объект: контекст блока
        со значением: начальный УИ.
      сам
       сохранить значение указателя стэка: количество аргументов
        в контекст: контекст блока.
      память
        сохранить указатель: Номер вызвавшего
```

```
        в объект: контекст блока
        со значением: активный контекст.
      сам новый активный контекст: контекст блока. ].
```

The primitiveValueWithArgs routine is associated with the valueWithArguments: messages in BlockContext. It is basically the same as the primitiveValue routine except that the block arguments come in a single Array argument to the valueWithArguments: message instead of as multiple arguments to the revalue"message.

элементарный значение с аргументами

```
 | аргумент ряд контекст блока количество аргументов блока класс ряда количество аргументов ряда начальный УИ |
  аргумент ряд ← сам вытолкнуть стэк.
  контекст блока ← сам вытолкнуть стэк.
 количество аргументов блока ← сам количество аргументов блока: контекст блока.
  класс ряда ← память извлечь класс: аргумент ряд.
  сам успех: класс ряда = Указатель на класс Ряд.
  сам успех
    истина: [
     количество аргументов ряда ← память извлечь длину в словах: аргумент ряд.
      сам
        успех: количество аргументов ряда
          = количество аргументов блока. ].
  сам успех
    истина: [
      сам
        перенести: количество аргументов ряда
        от номера: 0
        из объекта: аргумент ряд
        в номер: Начало блока временных
        в объект: контекст блока.
      начальный УИ ← память
        извлечь указатель: Номер начального УИ
        из объекта: контекст блока.
      память
```

```
            сохранить указатель: Номер указателя инструкции
              в объект: контекст блока
              со значением: начальный УИ.
            сам
              сохранить значение указателя стэка: количество аргументов ряда
              в контекст: контекст блока.
            память
              сохранить указатель: Номер вызвавшего
              в объект: контекст блока
              со значением: активный контекст.
            сам новый активный контекст: контекст блока. ]
          ложь: [сам отменить выталкивание: 2.].
```

The primitivePerform routine is associated with all perform"messages in Object (perform:, perform:with:, perform:with:with:, and so on). It is equivalent to sending a message to the receiver whose selector is the first argument of and whose arguments are the remaining arguments. It is, therefore, similar to the sendSelector:argumentCount: routine except that it must get rid of the selector from the stack before calling executeNewMethod and it must check that the CompiledMethod it finds takes one less argument that the "perform"message did, The primitive fails if the number of arguments does not match.

элементарный выполнить

```
  | выполняемый селектор новый получатель номер селектора |
  выполняемый селектор ← селектор сообщения.
   селектор сообщения ← сам значение стэка: количество аргументов − 1.
  новый получатель ← сам значение стэка: количество аргументов.
  сам
     искать метод в классе: ( память извлечь класс: новый получатель ).
  сам
    успех: ( сам количество аргументов: новый метод )
      = ( количество аргументов − 1 ).
  сам успех
    истина: [
```

```
        номер селектора ← указатель стэка − количество аргумен-
тов + 1.
      сам
        перенести: количество аргументов − 1
        от номера: номер селектора + 1
        из объекта: активный контекст
        в номер: номер селектора
        в объект: активный контекст.
      сам вытолкнуть: 1.
      количество аргументов ← количество аргументов − 1.
      сам выполнить новый метод.]
    ложь: [селектор сообщения ← выполняемый селектор.].
```

The primitivePerformWithArgs routine is associated with the perforrnWithArguments messages in Object. It is basically the same as the primitivePerform routine except that the message arguments come in a single Array argument to the performWithArguments: message instead of as multiple arguments to the perform"message.

элементарный выполнить с аргументами

```
  | этот получатель выполняемый селектор аргумент ряд класс ряда
размер ряда номер |
  аргумент ряд ← сам вытолкнуть стэк.
  размер ряда ← память извлечь длину в словах: аргумент ряд.
  класс ряда ← память извлечь класс: аргумент ряд.
  сам
    успех: указатель стэка + размер ряда
      < (память извлечь длину в словах: активный контекст).
  сам успех: класс ряда = Указатель на класс Ряд.
  сам успех
    истина: [
      выполняемый селектор ← селектор сообщения.
      селектор сообщения ← сам вытолкнуть стэк.
      этот получатель ← сам вершина стэка.
      количество аргументов ← размер ряда.
      номер ← 1.
      [номер <= количество аргументов.]
        пока истина: [
```

```
            сам
              протолкнуть: ( память
                извлечь указатель: номер − 1
                из объекта: аргумент ряд ).
            номер ← номер + 1. ].
        сам
           искать метод в классе: ( память извлечь класс: этот получатель ).
        сам
          успех: ( сам количество аргументов: новый метод )
            = количество аргументов.
        сам успех
          истина: [ сам выполнить новый метод. ]
          ложь: [
            сам
              отменить выталкивание: количество аргументов.
            сам протолкнуть: селектор сообщения.
            сам протолкнуть: аргумент ряд.
            количество аргументов ← 2.
            селектор сообщения ← выполняемый селектор. ]. ]
      ложь: [ сам отменить выталкивание: 1. ].
```

The next four primitive routines (for primitive indices 85 through 88) are used for communication and scheduling of independent processes. The following routine initializes the indices used to access Processes, ProcessorSchedulers, and Semaphores.

инициализировать номера Планировщика

```
Номер списка процессов ← 0.
Номер активного процесса ← 1.
Номер первой связи ← 0.
Номер последней связи ← 1.
Номер избыточных сигналов ← 2.
Номер следующей связи ← 0.
Номер приостановленного контекста ← 1.
Номер приоритета ← 2.
Номер моего списка ← 3.
```

Process switching must be synchronized with the execution of bytecodes. This is done using the following four interpreter registers and the four routines: checkProcessSwitch, asynchronousSignal:, synchronousSignal:, and transferTo:.

Регистры интерпретатора связанные с процессами

новый процесс ждёт	The newProcessWaiting register will be true if a process switch is called for and false otherwise.
новый процесс	If newProcessWaiting is true then the newProcess register will point to the Process to be transferred to.
список семафоров	The semaphoreList register points to an Array used by the interpreter to buffer Semaphores that should be signaled. This is an Array in Interpreter, not in the object memory. It will be a table in a machine-language interpreter.
номер семафора	The semaphoreIndex register holds the index of the last Semaphore in the semaphoreList buffer.

The asynchronousSignal: routine adds a Semaphore to the buffer.

```
асинхронный сигнал: семафор
  номер семафора ← номер семафора + 1.
  список семафоров от: номер семафора пом: семафор.
```

The Semaphores will actually be signaled in the checkProcessSwitch routine which calls the synchronousSignal: routine once for each Semaphore in the buffer. If a Process is waiting for the Semaphore, the synchronousSignal: routine resumes it. If no Process is waiting, the synchronousSignal: routine increments the Semaphore's count of excess signals. The isEmptyList:, resume:, and removeFirstLinkOfList: routines aredescribed later in this section.

```
синхронный сигнал: семафор
  | избыточные сигналы |
  ( сам это пусто список: семафор )
    истина: [
```

```
        избыточные сигналы ← сам
          достать целое: Номер избыточных сигналов
          из объекта: семафор.
        сам
          сохранить целое: Номер избыточных сигналов
          в объект: семафор
          со значением: избыточные сигналы + 1. ]
      ложь: [
        сам
          resume: ( сам удалить первую связь списка: семафор ). ].
```

The transferTo: routine is used whenever the need to switch processes is detected. It sets the newProcessWaiting and newProcess registers.

```
перейти к: процесс
  новый процесс ждёт ← истина.
  новый процесс ← процесс.
```

The checkProcessSwitch routine is called before each bytecode fetch (in the interpret routine) and performs the actual process switch if one has been called for. It stores the active context pointer in the old Process, stores the new Process in the ProcessorScheduler's active process field, and loads the new active context out of that Process.

```
проверить переключение процессов
  | активный процесс |
  [ номер семафора > 0. ]
    пока истина: [
      сам
      синхронный сигнал: ( список семафоров от: номер семафора ).
      номер семафора ← номер семафора − 1. ].
  новый процесс ждёт
    истина: [
      новый процесс ждёт ← ложь.
      активный процесс ← сам активный процесс.
      память
        сохранить указатель: Номер приостановленного контекста
        в объект: активный процесс
        со значением: активный контекст.
```

```
        память
            сохранить указатель: Номер активного процесса
            в объект: сам указатель на Планировщика
            со значением: новый процесс.
        сам
            новый активный контекст: ( память
                извлечь указатель: Номер приостановленного контекста
                из объекта: новый процесс ). ].
```

Any routines desiring to know what the active process will be must take into account the newerocessWaiting and newerocess registers. Therefore, they use the following routine.

активный процесс

```
    новый процесс ждёт
        истина: [↑ новый процесс. ]
        ложь: [
            ↑ память
                извлечь указатель: Номер активного процесса
                из объекта: сам указатель на Планировщика. ].
```

The instance of ProcessorScheduler responsible for scheduling the actual processor needs to be known globally so that the primitives will know where to resume and suspend Processes. This ProcessorScheduler is bound to the name Processor in the Smalltalk global dictionary. The association corresponding to Processor has a guaranteed object pointer, so the appropriate ProcessorScheduler can be found.

указатель на Планировщика

```
    ↑ память
        извлечь указатель: Номер значения
        из объекта: Указатель на Ассоциацию планировщика.
```

When Smalltalk is started up, the initial active context is found through the scheduler's active Process.

первый контекст

```
    новый процесс ждёт ← ложь.
    ↑ память
        извлечь указатель: Номер приостановленного контекста
        из объекта: сам активный процесс.
```

If the object memory automatically deallocates objects on the basis of reference counting, special consideration must be given to reference counting in the process scheduling routines. During the execution of some of these routines, there will be times at which there are no references to some object from the object memory (e.g., after a Process has been removed from a Semaphore but before it hasbeen placed on one of the ProcessorScheduler's LinkedLists). If the object memory uses garbage collection, it simply must avoid doing a collection in the middle of a primitive routine. The routines listed here ignore the reference-counting problem in the interest of clarity. Implementations using reference counting will have to modify these routines in order to prevent premature deallocation of objects.

The following three routines are used to manipulate LinkedLists.

```
удалить первую связь списка: связанный список
  | первая связь последняя связь следующая связь |
  первая связь ← память
    извлечь указатель: Номер первой связи
    из объекта: связанный список.
  последняя связь ← память
    извлечь указатель: Номер последней связи
    из объекта: связанный список.
  последняя связь = первая связь
    истина: [
      память
        сохранить указатель: Номер первой связи
        в объект: связанный список
        со значением: Указатель на пусто.
      память
        сохранить указатель: Номер последней связи
        в объект: связанный список
        со значением: Указатель на пусто. ]
    ложь: [
      следующая связь ← память
        извлечь указатель: Номер следующей связи
        из объекта: первая связь.
      память
        сохранить указатель: Номер первой связи
```

```
            в объект: связанный список
            со значением: следующая связь. ].
    память
        сохранить указатель: Номер следующей связи
        в объект: первая связь
        со значением: Указатель на пусто.
    ↑первая связь.

добавить последней связью: связь
к списку: связанный список
    | последняя связь |
    ( сам это пустой список: связанный список )
        истина: [
            память
                сохранить указатель: Номер первой связи
                в объект: связанный список
                со значением: связь. ]
        ложь: [
            последняя связь ← память
                извлечь указатель: Номер последней связи
                из объекта: связанный список.
            память
                сохранить указатель: Номер следующей связи
                в объект: последняя связь
                со значением: связь. ].
    память
        сохранить указатель: Номер последней связи
        в объект: связанный список
        со значением: связь.
    память
        сохранить указатель: Номер моего списка
        в объект: связь
        со значением: связанный список.

это пустой список: связанный список
    ↑( память
        извлечь указатель: Номер первой связи
```

```
    из объекта: связанный список )
      = Указатель на пусто.
```

These three LinkedList routines are used, in turn, to implement the following two routines that remove links from or add links to the ProcessorScheduler's LinkedLists of quiescent Processes.

```
разбудить процесс с наивысшим приоритетом
  | приоритет списки процессов список процессов |
  списки процессов ← память
    извлечь указатель: Номер списка процессов
    из объекта: сам указатель на Планировщика.
  приоритет ← память извлечь длину в словах: списки процессов.
  [
    список процессов ← память
      извлечь указатель: приоритет − 1
      из объекта: списки процессов.
    сам это пустой список: список процессов. ]
      пока истина: [ приоритет ← приоритет − 1. ].
  ↑ сам удалить первую связь списка: список процессов.

сон: процесс
  | приоритет списки процессов список процессов |
   приоритет ← сам достать целое: Номер приоритета из объекта:
процесс.
  списки процессов ← память
    извлечь указатель: Номер списка процессов
    из объекта: сам указатель на Планировщика.
  список процессов ← память
    извлечь указатель: приоритет − 1
    из объекта: списки процессов.
  сам
    добавить последней связью: процесс
    к списку: список процессов.
```

These two routines are used, in turn, to implement the following two routines that actually suspend and resume Processes.

```
приостановить активный
  сам
```

```
    перейти к: сам разбудить процесс с наивысшим приоритетом.

возобновить: процесс
  | активный процесс приоритет активного новый приоритет |
  активный процесс ← сам активный процесс.
  приоритет активного ← сам
    достать целое: Номер приоритета
    из объекта: активный процесс.
  новый приоритет ← сам достать целое: Номер приоритета из объ-
екта: процесс.
  новый приоритет > приоритет активного
    истина: [сам сон: активный процесс. сам перейти к: процесс.]
    ложь: [сам сон: процесс.].
```

The primitiveSignal routine is associated with the signal message in Semaphore. Since it is called in the process of interpreting a bytecode, it can use the synchronousSignal: routine. Any other signaling of Semaphores by the interpreter (for example, for timeouts and keystrokes) must use the asynchronousSignal: routine.

```
элементарный сигнал
  сам синхронный сигнал: сам вершина стэка.
```

The primitiveWait routine is associated, with the wait message in Semaphore. If the receiver has an excess signal count greater than O, the primitiveWait routine decrements the count. If the excess signal count is O, the primitiveWait suspends the active Process and adds it to the receiver's list of Processes.

```
элементарный ждать
  | этот получатель избыточные сигналы |
  этот получатель ← сам вершина стэка.
  избыточные сигналы ← сам
    достать целое: Номер избыточных сигналов
    из объекта: этот получатель.
  избыточные сигналы > 0
    истина: [
      сам
        сохранить целое: Номер избыточных сигналов
        в объект: этот получатель
```

```
            со значением: избыточные сигналы — 1. ]
        ложь: [
            сам
                добавить последней связью: сам активный процесс
                к списку: этот получатель.
            сам приостановить активный. ].
```

The primitiveResume routine is associated with the resume message in Process. It simply calls the resume: routine with the receiver as argument.

элементарный возобновить

```
сам возобновить: сам вершина стэка.
```

The primitiveSuspend routine is associated with the suspend message in Process. The primitiveSuspend routine suspends the receiver if it is the active Process. If the receiver is not the active Process, the primitive fails.

элементарный приостановить

```
сам успех: сам вершина стэка = сам активный процесс.
сам успех
    истина: [
        сам вытолкнуть стэк.
        сам протолкнуть: Указатель на пусто.
        сам приостановить активный. ].
```

The primitiveFlushCache routine removes the contents of the method cache. Implementations that do not use a method cache can treat this as a no-op.

элементарный очистить кэш

```
сам инициализировать кэш методов.
```

29.5 Элементарные методы ввода-вывода

The input/output primitive routines provide Smalltalk with access to the state of the hardware devices. Since the implementation of these routines will be dependent on the structure of the implementing machine,

no routines will be given, just a specification of the behavior of the primitives.

```
выполнить элементарный метод ввода вывода
    номер элементарного метода = 90
        истина: [↑сам элементарный точка мыши.].
    номер элементарного метода = 91
        истина: [↑сам элементарный поместить курсор в положение.].
    номер элементарного метода = 92
        истина: [↑сам элементарный привязать курсор.].
    номер элементарного метода = 93
        истина: [↑сам элементарный семафор ввода.].
    номер элементарного метода = 94
        истина: [↑сам элементарный интервал семафора.].
    номер элементарного метода = 95
        истина: [↑сам элементарный ввести слово.].
    номер элементарного метода = 96
        истина: [↑сам элементарный копировать биты.].
    номер элементарного метода = 97
        истина: [↑сам элементарный сделать снимок.].
    номер элементарного метода = 98
        истина: [↑сам элементарный слова времени в.].
    номер элементарного метода = 99
        истина: [↑сам элементарный слова тиков в.].
    номер элементарного метода = 100
        истина: [↑сам элементарный сигналить на тике.].
    номер элементарного метода = 101
        истина: [↑сам элементарный стать курсором.].
    номер элементарного метода = 102
        истина: [↑сам элементарный стать экраном.].
    номер элементарного метода = 103
        истина: [↑сам элементарный просмотреть знаки.].
    номер элементарного метода = 104
        истина: [↑сам элементарный цикл отрисовки.].
    номер элементарного метода = 105
        истина: [↑сам элементарный заменить цепь.].
```

Four of the primitive routines are used to detect actions by the user. The two types of user action the system can detect are changing the state of a bi-state device and moving the pointing device. The bi-state devices are the keys on the keyboard, three buttons associated with the pointing device and an optional five-paddle keyset. The buttons associated with the pointing device may or may not actually be on the physical pointing device. Three of the four input primitive routines (primitiveInputSemaphore, primitiveInputWord, and primitiveSampleInterval) provide an active or event-initiated mechanism to detect either state change or movement. The other primitive routine (primitiveMousePoint) provides a passive or polling mechanism to detect pointing device location.

The event-initiated mechanism provides a buffered stream of 16-bit words that encode changes to the bi-state devices or the pointing device location. Each time a word is placed in the buffer, a Semaphore is signaled (using the asynchronousSignal: routine). The Semaphore to signal is initialized by the primitiveInputSemaphore routine. This routine is associated with the primInputSemaphore: message in InputState and the argument of the message becomes the Semaphore to be signaled. The primitiveInputWord routine (associated with the primInputWord message in InputState) returns the next word in the buffer, removing it from the buffer. Since the Semaphore is signaled once for every word in the buffer, the Smalltalk process emptying the buffer should send the Semaphore a wait message before sending each primInputWord message. There are six types of 16-bit word placed in the buffer. Two types specify the time of an event, two types specify state change of a bi-state device, and two types specify pointing device movement. The type of the word is stored in its high order four bits. The low order 12-bits are referred to as the parameter.

The six type codes have the following meanings.

код типа	*значение*
0	Delta time (the parameter is the number of milliseconds since the last event of any type)
1	X location of the pointing device
2	Y location of the pointing device
3	Bi-state device turned on (the parameter indicates which device)

4	Bi-state device turned off (the parameter indicates which device)
5	Absolute time (the parameter is ignored, the next two words in the buffer contain a 32-bit unsigned number that is the absolute value of the millisecond clock)

Whenever a device state changes or the pointing device moves, a time word is put into the buffer. A type 0 word will be used if the number of milliseconds since the last event can be represented in 12 bits. Otherwise, a type 5 event is used followed by two words representing the absolute time. Note that the Semaphore will be signaled 3 times in the latter case. Following the time word(s) will be one or more words of type 1 through 4. Type 1 and 2 words will be generated whenever the pointing device moves at all. It should be remembered that Smalltalk uses a left-hand coordinate system to talk about the screen. The origin is the upper left corner of the screen, the x dimension increases toward the right, and the y dimension increases toward the bottom. The minimum time span between these events can be set by the primitiveSamplelntervat routine which is associated with the primSamplelnterval: message in lnputState. The argument to primSamplelnterval: specifies the number of milliseconds between movement events if the pointing device is moving constantly.

Type 3 and 4 words use the low-order eight bits of the parameter to specify which device changed state. The numbering scheme is set up to work with both decoded and undecoded keyboards. An undecoded keyboard is made up of independent keys with independent down and up transitions. A decoded keyboard consists of some independent keys and some meta"keys (shift and escape) that cannot be detected on their own, but that change the value of the other keys. The keys on a decoded keyboard only indicate their down transition, not their up transition. On an undecoded keyboard, the standard keys produce parameters that are the ASCII code of the character on the keytop without shift or control information (i.e,, the key with A"on it produces the ASCII for ' a"and the key with 2"and ??@"on it produces the ASCII for 2"). The other standard keys produce the following parameters.

key	*parameter*

backspace	8
tab	9
line feed	10
return	13
escape	27
space	32
delete	127

For an undecoded keyboard, the meta keys have the following parameters.

key	*parameter*
left shift	136
right shift	137
control	138
alpha-lock	139

For a decoded keyboard, the full shifted and controlled"ASCII should be used as a parameter and successive type 3 and 4 words should be produced for each keystroke.

The remaining bi-state devices have the following parameters.

key	parameter
left or top «pointing device» button	128
center «pointing device» button	129
right or bottom «pointing device» button	130
keyset paddles right to left	131 through 135

The primitiveMousePoint routine allows the location of the pointing device to be polled. It allocates a new Point and stores the location of the pointing device in its x and y fields.

The display screen is a rectangular set of pixels that can each be one of two colors. The colors of the pixels are determined by the individual bits in a specially designated instance of DisplayScreen. DisplayScreen is a subclass of Form. The instance of DisplayScreen that should be used to update the screen is designated by sending it the message beDisplay. This message invokes the primitiveBeDisplay primitive routine. The screen will be updated from the last recipient of beDisplay approximately 60

times a second.

Every time the screen is updated, a cursor is ORed into its pixels. The cursor image is determined by a specially designated instance of Cursor. Cursor is a subclass of Form whose instances always have both width and height of 16. The instance of Cursor that should be ORed into the screen is designated by sending it the message beCursor. This message invokes the primitiveBeCursor primitive routine.

The location at which the cursor image should appear is called the cursor location. The cursor location may be linked to the location of the pointing device or the two locations may be independent. Whether or not the two locations are linked is determined by sending a message to class Cursor with the selector cursorLink: and either true or false as the argument. If the argument is true, then the two locations will be the same; if it is false, they are independent. The cursorLink: message in Cursor's metaclass invokes the primitiveCursorLink primitive routine.

The cursor can be moved in two ways. If the cursor and pointing device have been linked, then moving the pointing device moves the cursor. The cursor can also be moved by sending the message primCursorLocPut: to an instance of lnputState. This message takes a Point as an argument and invokes the primitiveCursorLocPut primitive routine. This routine moves the cursor to the location specified by the argument. If the cursor and pointing device are linked, the primitiveCursorLocPut routine also changes the location indicated by the pointing device.

The primitiveCopyBits routine is associated with the copyBits message in BitBIt and performs an operation on a bitmap specified by the receiver. This routine is described in Chapter 18.

The primitiveSnapshot routine writes the current state of the object memory on a file of the same format as the Smalltalk-80 release file. This file can be resumed in exactly the same way that the release file was originally started. Note that the pointer of the active context at the time of the primitive call must be stored in the active Process on the file.

The primitiveTimeWordslnto and primitiveTickWordslnto routines are associated with the timeWordslnto: and tickWordsinto: messages in Sensor. Both of these messages take a byte indexable object of at least four bytes as an argument. The primitiveTimeWordslnto routine stores the number of seconds since the midnight previous to January 1, 1901

as an unsigned 32-bit integer into the first four bytes of the argument. The primitiveTickWordsInto routine stores the number of ticks of the millisecond clock (since it last was reset or rolled over) as an unsigned 32-bit integer into the first four bytes of the argument.

The primitiveSignalAtTick routine is associated with the siflnal:atTick: messages in ProcessorScheduler. This message takes a Semaphore as the first argument and a byte indexable object of at least four bytes as the second argument. The first four bytes of the second argument are interpreted as an unsigned 32-bit integer of the type stored by the primitiveTickWordsInto routine. The interpreter should signal the Semaphore argument when the millisecond clock reaches the value specified by the second argument. If the specified time has passed, the Semaphore issignaled immediately. This primitive signals the last Semaphore to be passed to it. If a new call is made on it before the last timer value has been reached, the last Semaphore will not be signaled. If the first argument is not a Semaphore, any currently waiting Semaphore will be forgotten.

The primitiveScanCharacters routine is an optional primitive associated with the scanCharactersFrom:to:in:rightX:stopConditions:displaying message in CharacterScanner. If the function of the Smalltalk method is duplicated in the primitive routine, text display will go faster. The primitiveDrawLoop routine is similarly an optional primitive associated with the drawLoopX:Y: message in BitBIt. If the function of the Smalltalk method is duplicated in the primitive routine, drawing lines will go faster.

29.6 Элементарные методы системы

The seven final primitives are grouped together as system primitives.

```
выполнить элементарный метод системы
  номер элементарного метода = 110
    истина: [↑сам элементарный эквивалентен.].
  номер элементарного метода = 111
    истина: [↑сам элементарный метод класс.].
  номер элементарного метода = 112
    истина: [↑сам элементарный оставшаяся память.].
  номер элементарного метода = 113
```

```
    истина: [↑сам элементарный выйти.].
  номер элементарного метода = 114
    истина: [↑сам элементарный выйти в отладчик.].
  номер элементарного метода = 115
    истина: [↑сам элементарный оставшиеся УО.].
  номер элементарного метода = 116
    истина: [
      ↑сам
        элементарный сигналить при оставшихся УО при оставших-
ся словах.].
```

The primitiveEquivalent routine is associated with the = = message in Object. It returns true if the receiver and argument are the same object (have the same object pointer) and false otherwise.

элементарный эквивалентен

```
  | этот объект тот объект |
  тот объект ← сам вытолкнуть стэк.
  этот объект ← сам вытолкнуть стэк.
  этот объект = тот объект
    истина: [сам протолкнуть: Указатель на истину.]
    ложь: [сам протолкнуть: Указатель на ложь.].
```

The primitiveClass routine is associated with the class message in Object. It returns the object pointer of the receiver's class.

элементарный метод класс

```
  | экземпляр |
  экземпляр ← сам вытолкнуть стэк.
  сам протолкнуть: (память извлечь класс: экземпляр).
```

The primitiveCoreLeft routine returns the number of unallocated words in the object space. The primitiveQuit routine exits to another operating system for the host machine, if one exists. The primitiveExitToDebugger routine calls the machine language debugger, if one exists.

Глава 30

Формальное определение памяти объектов

Часть V

Словари

Глава 31

Псевдо переменные

true	истина
false	ложь
nil	пусто
self	сам
super	над
thisContext	этот контекст

Глава 32

Словари селекторов

Оглавление

32.1 Английско-русский словарь селекторов

A

ArrayConstant	Ряд констант
abs	модуль
accessingSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:	подкласс доступа: имена переменных экземпляра: имена переменных класса: словари пула: категория:
activateNewMethod	активировать новый метод
activePriority	активный приоритет
activeProcess	активный процесс
add:	добавить:
add: after:	добавить: после:

add: before:	добавить: перед:
add: withOccurrences:	добавить: с вхождениями:
addAll:	добавить все:
addAllFirst:	добавить первыми все:
addAllLast:	добавить последними все:
addArguments:	добавить аргументы:
addArguments: andTemporaryVariables:	добавить аргументы: и временные переменные:
addAssociationForEnvironmentVariable: inClass:	добавить ассоциацию для переменной окружения: в классе:
addAssociationForGlobalVariable:	добавить ассоциацию для глобальной переменной:
addAssociationForGlobalVariable: inClass:	добавить ассоциацию для глобальной переменной: в классе:
addClassVarName:	добавить имя переменной класса:
addDays:	добавить дни:
addDependent:	добавить зависимость:
addFirst:	добавить первым:
addInstVarName:	добавить имя переменной экземпляра:
addLast:	добавить последним:
addLastLink: toList:	добавить последней связью: к списку:
addLiteral:	добавить литерал:
addLiteral: node:	добавить литерал: узел:
addLiteralsTo:	добавить литералы к:
addSelector: withMethod:	добавить селектор: с методом:
addSharedPool:	добавить разделяемый пул:
addStyle:	добавить стиль:
addTemporaryVariable:	добавить временную переменную:
addTemporaryVariables:	добавить временные переменные:
addTime:	добавить время:
after:	после:

allClassVarNames	все имена переменных класса
allInstances	все экземпляры
allInstancesDo:	делать для всех экземпляров:
allInstVarNames	все имена переменных экземпляра
allMask:	вся маска:
allSelectors	все селекторы
allSharedPools	все разделяемые пулы
allSubclasses	все подклассы
allSubclassesDo:	делать для всех подклассов:
allSubInstancesDo:	делать для все подэкземпляров:
allSuperclasses	все надклассы
allSuperclassesDo:	делать для всех надклассов:
amountTempVars	количество врем пер
anchor	якорь
and:	и:
and: and:	и: и:
anyMask:	любой из маски:
anySatisfy:	любой удовлетворяет:
apply:	применить:
argumentCountOf:	количество аргументов:
argumentCountOfBlock:	количество аргументов блока:
argumentNames	имена аргументов
argumentPrecedence:	приоритет аргумента:
arguments	аргументы
arguments:	аргументы:
arguments: temporaries: statements:	аргументы: временные: предложения:
argumentsNumber	количество аргументов
arithmeticSelectorPrimitive	элементарный метод арифметический селектор
as:	как:
asArray	как ряд
asBag	как мешок
asByteArray	как ряд байтов
asCharacter	как знак

asciiValue	значение АСКОИ
asFloat	как плавающее
asFraction	как дробь
asInteger	как целое
asLowercase	в нижнем регистре
asObject	как объект
asOrderedCollection	как упорядоченный набор
asOriginalLiteral	как оригинальный литерал
asSeconds	как секунды
asSet	как множество
asSortedCollection	как сортированный набор
asSortedCollection:	как сортированный набор:
asStatement	как предложение
asString	как цепь
asSymbol	как символ
assert:	утверждение:
associationAt:	ассоциация от:
associationAt: ifAbsent:	ассоциация от: если нету:
associationForClassVariable: inClass:	ассоциация для переменной класса: в классе:
associationsDo:	ассоциации делать:
asUppercase	в верхнем регистре
asynchronousSignal:	асинхронный сигнал:
at:	от:
at: ifAbsent:	от: если нету:
at: ifPresent:	от: если есть:
at: put:	от: пом:
atAll: put:	от всех: пом:
atAll: putAll:	от всех: пом все:
atAllPut:	от всех пом:
atEnd	в конце
atNewIndex: put:	от нового номера: пом:
attributeAt:	атрибут от:
attributeAt: put:	атрибут от: пом:
attributes	атрибуты
attributes:	атрибуты:

B

basicAt:	основной от:
basicAt: put:	основной от: пом:
basicNew	основной новый
basicNew:	основной новый:
basicSize	основной размер
become:	становится:
before:	перед:
beginsWith:	начинается с:
between: and:	между: и:
bindWith:	связать с:
bitAnd:	побитовое и:
bitInvert	обратить биты
bitOr:	побитовое или:
bitShift:	сдвинуть биты:
bitXor:	побитовое искл или:
block	блок
block:	блок:
blockArgument	аргумент блока
blockCopy	экземпляр блока
blockCopy:	экземпляр блока:
blockquote	блок цитата
blockquote:	блок цитата:
braceArray	фигурный ряд
braceStream:	фигурный поток:
break	разрыв
brush:	кисть:
bytecodesFor:	байткоды для:

C

Character	Знак
caller	вызвавший
cannotReturn:	невозможно вернуть:
canUnderstand:	может понимать:

cashOnHand	количество наличных
category	категория
category:	категория:
ceiling	потолок
center	центр
changed	изменён
changeInsurenceLimit:	изменить предел страховки:
changeLanguage	сменить язык
changeTranslationFor: on:	изменить перевод для: на:
checkIndexableBoundsOf: in:	проверить границы нумерования для: в:
checkInstanceVariableBoundsOf: in:	проверить границы переменных экземпляра для: в:
checkProcessSwitch	проверить переключение процессов
class	класс
class:	класс:
class: if:	класс: если:
classes	классы
classPool	пул класса
classVariableNames:	имена переменных класса:
classVariablesArrayIn:	ряд переменных класса на:
classVariablesString	цепь переменных класса
classVariablesStringIn:	цепь переменных класса на:
classVarNames	имена переменных класса
clear	очистить
close	закрыть
code	код
code:	код:
collect:	собрать:
collection: map:	набор: карта:
colorPrintArgumentsOn: ident: inOneLine:	цветная печать аргументов в: отступ: в одну строку:
colorPrintOn:	цветная печать в:
colorPrintOn: ident:	цветная печать в: отступ:
colorPrintOn: ident: inOneLine:	цветная печать в: отступ: в одну строку:

colorPrintOn: templateForTranslateVariablesIn: dictionaryName:	цветная печать в: шаблона для перевода переменных на: имя словарь:
colorPrintStatementsOn: ident: inOneLine:	цветная печать предложений в: отступ: в одну строку:
colorPrintTemporariesOn: ident: inOneLine:	цветная печать временных в: отступ: в одну строку:
comment	комментарий
comment:	комментарий:
commonSelectorPrimitive	элементарный метод общий селектор
compile:	компилировать:
compile: classified:	компилировать: классифицировать:
compile: classified: notifying:	компилировать: классифицировать: уведомлять:
compile: notifying:	компилировать: уведомлять:
compileAll	компилировать весь
compiledMethodAt:	откомпилированный метод от:
containsInstanceOf:	содержит экземпляр:
containsPoint:	содержит точку:
contents	содержимое
contents:	содержимое:
contentsClass	класс содержимого
convertToDoIt	преобразовать для выполнения
copy	копия
copy: from:	копировать: из:
copy: from: classified:	копировать: из: классифицировать:
copyAll: from:	копировать все: из:
copyAll: from: classified:	копировать все: из: классифицировать:
copyAllCategoriesFrom:	копировать все категории из:
copyCategory: from:	копировать категорию: из:
copyCategory: from: classified:	копировать категорию: из: классифицировать:

copyEmpty	пустая копия
copyFrom: to:	копия от: до:
copyReplaceAll: with:	копия с заменой всех: на:
copyReplaceFrom: to: with:	копия с заменой от: до: на:
copyWith:	копия с:
copyWithout:	копия без:
corner	угол
corner:	угол:
count:	количество:
cr	пс
createActualMessage	создать текущее сообщение
critical:	критический:
crtab	пс таб
crtab:	пс таб:
cycle	цикл

D

data:	данные:
dateAndTimeNow	текущие дата и время
dayOfWeek:	день недели:
daysInMonth: forYear:	дней в месяце: для года:
daysInYear:	дней в году:
decompile:	декомпилировать:
decreaseReferencesTo:	уменьшить ссылки на:
deepCopy	глубокая копия
defaultBrowserTitle	заголовок браузера по умолчанию
defaultCompilerLanguage	язык компилятора по умолчанию
definitionData	данные определения
definitionData:	данные определения:
definitionList	список определений
definitionList:	список определений:
definitionTerm	определяемый термин
definitionTerm:	определяемый термин:

degreesToRadians	градусы в радианы
dependents	зависимости
detect:	выявить:
detect: ifNone:	выявить: если ни одного:
dictionary	словарь
digitAt:	цифра от:
digitAt: put:	цифра от: пом:
digitLength	длина цифр
dispatchArithmeticPrimitives	выполнить арифметический элементарный метод
dispatchControtPrimitives	выполнить управляющий элементарный метод
dispatchFIoatPrimitives	выполнить элементарный метод плавающего
dispatchIntegerPrimitives	выполнить элементарный метод целого
dispatchLargeIntegerPrimitives	выполнить элементарный метод большого целого
dispatchlnputOutputPrimitives	выполнить элементарный метод ввода вывода
dispatchOnThisBytecode	выполнить этот байткод
dispatchPrimitives	выполнить элементарный метод
dispatchPrivatePrimitives	выполнить собственный элементарный метод
dispatchStorageManagement-Primitives	выполнить элементарный метод управления хранилищем
dispatchSubscriptAndStream-Primitives	выполнить элементарные метоы нумерации и Потоков
dispatchSystemPrimitives	выполнить элементарный метод системы
display	показать
displayAt:	показать в:
div	раздел
div:	раздел:
do:	делать:
do: separatedBy:	делать: с разделителем:

document	документ
document:	документ:
doesNotUnderstand:	не понимаю:
doubleExtendedSendBytecode	байткод посылки с двумя расширениями
doubleExtendedSuperBytecode	байткод посылки наду с двумя расширениями
duplicateTopBytecode	байткод удвоить вершину

E

eighth	восьмой
element	элемент
elements:	элементы:
emitBytecodesOn:	вывести байткоды в:
emitExceptLastOn: with:	вывести без последнего в: с:
emitForEffectOn: with:	вывести код для эффекта в: с:
emitForEffectOn: with: sendTo:	вывести код для эффекта в: с: посылать к:
emitForEvaluatedValueOn: with:	вывести код для выполнения в: с:
emitForValueOn: with:	вывести код для значения в: с:
emitForValueOn: with: sendTo:	вывести код для значения в: с: посылать к:
emitStoreOn: with:	вывести код для запоминания в: с:
emitStorePopOn: with:	вывести код для запоминания и извлечения в: с:
emptyCheck	проверить на пустость
endsWith:	заканчивается на:
ensure:	гарантировать:
environment	окружение
eqv:	экв:
error	ошибка
error:	ошибка:
errorKeyNotFound	ошибка ключ не найден

errorNoSuchElement	ошибка нету такого элемента
errorNotFound:	ошибка не найден:
errorNotKeyed	ошибка не ключевой
errorSubscriptBounds:	ошибка границы номера:
errorValueNotFound	ошибка значение не найдено
even	чётное
executeNewMethod	выполнить новый метод
exp	эксп
exponent	экспонента
expression:	выражение:
expression: level:	выражение: уровень:
extendedPushBytecode	байткод расширенное помещение
extendedSendBytecode	расширенный байткод посылки
extendedStoreAndPopBytecode	расширенный байткод сохранить и вытолкнуть
extendedStoreBytecode	расширенный байткод сохранить
extent	размеры
extent:	размеры:
extractBits: to: of:	извлечь биты от: до: из:

F

factorial	факториал
false	ложь
fetchByte	извлечь байт
fetchByte: ofObject:	извлечь байт: из объекта:
fetchByteLengthOf:	извлечь длину в байтах:
fetchClassOf:	извлечь класс:
fetchContextRegisters	извлечь регистры контекста
fetchInteger: ofObject:	достать целое: из объекта:
fetchPointer: ofObject:	извлечь указатель: из объекта:
fetchWord: ofObject:	извлечь слово: из объекта:
fetchWordLengthOf:	извлечь длину в словах:

fieldIndexOf:	номер поля для:
fifth	пятый
fileNamed:	имя файла:
fileOut	вывести в файл
fileOutCategory:	вывести в файл категорию:
fileOutChangedMessages: on:	вывести в файл изменённые сообщения: в:
fileOutOn:	вывести в файл:
findElementOrNil:	найти элемент или пусто:
findFirst:	найти первый:
findLast:	найти последний:
findNewMethodInClass:	найти новый метод в классе:
first	первый
firstContext	первый контекст
fixCollisionsFrom:	устранить конфликт в:
fixedFieldsOf:	фиксированных полей:
fixTemps	настроить временные
flagValueOf:	значение флага:
floor	пол
for:	для:
forAllMethodVariablesDo:	для всех переменных метода делать:
forAllSymbolsDo:	для всех символов делать:
fork	разветвить
forkAt:	разветвить от:
forMilliseconds:	на миллисекунды:
forMutualExclusion	для взаимного исключения
form	форма
form:	форма:
format:	формат:
forSeconds:	на секунды:
fourth	четвёртый
fractionPart	дробная часть
from:	из:
from: to:	от: до:
from: to: by:	от: до: через:
from: value:	из: значение:

fromDays:	из дней:
fromSeconds:	из секунд:
fromString:	из цепи:

G

gcd:	нод:
grow	расти

H

halt	останов
hash	хэш
hash:	хэш:
hasMethods	имеет методы
hasObject:	содержит объект:
headerExtensionOf:	расширение заголовка:
heading	заголовок
heading:	заголовок:
height	высота
highBit	старший бит
highByteOf:	старший байт:
highIOPriority	высокий приоритет ВВ
horizontalRule	горизонтальная линия
hours	часы
html:	яргт:

I

IongConditionalJump	длинный условный прыжок
ifEmpty:	если пустой:
ifEmpty: ifNotEmpty:	если пустой: если не пустой:
ifFalse:	ложь:
ifFalse: ifTrue:	ложь: истина:

ifNil:	пусто:
ifNil: ifNotNil:	пусто: не пусто:
ifNotEmpty:	если не пустой:
ifNotEmpty: ifEmpty:	если не пустой: если пустой:
ifNotNil:	не пусто:
ifNotNil: ifNil:	не пусто: пусто:
ifTrue:	истина:
ifTrue: ifFalse:	истина: ложь:
image	изображение
image:	изображение:
incAmountTempVars	увеличить колво врем пер
includeR	содержит о
includes:	содержит:
includesAssociation:	содержит ассоциацию:
includesKey:	содержит ключ:
includesSelector:	содержит селектор:
increaseReferencesTo:	увеличить ссылки на:
index:	номер:
indexOf:	номер для:
indexOf: ifAbsent:	номер для: если нету:
indexOfExponent	номер экспоненты
indexOfGlobalVariable:	номер глобальной переменной:
indexOfLiteral:	номер литерала:
indexOfMonth:	номер месяца:
indexOfRadix	номер основания
indexOfSubCollection: startingAt:	номер поднабора: начиная с:
indexOfSubCollection: startingAt: ifAbsent:	номер поднабора: начиная с: если нету:
inheritsFrom:	наследует от:
initialBalance:	инициализировать баланс:
initialInstanceOf:	начальный экземпляр для:
initialInstructionPointerOfMethod:	начальный указатель инструкции метода:
initialize	инициализировать
initialize:	инициализировать:

initializeAssociationIndex	инициализировать номер Ассоциации
initializeCharacterIndex	инициализировать номера Знака
initializeClassIndices	инициализировать номера класса
initializeContextIndices	инициализировать номера контекста
initializeDeductions	инициализировать налоги
initializeExponentsDictionary	инициализировать словарь экспонент
initializeMessageIndices	инициализировать номера Сообщения
initializeMethodCache	инициализировать кэш методов
initializePointIndices	инициализировать номера Точки
initializeRadixDictionary	инициализировать словарь оснований
initializeSchedulerIndices	инициализировать номера Планировщика
initializeSpecialNames	инициализировать специальные имена
initializeStreamIndices	инициализировать номера Потока
initializeTimingProcess	инициализировать синхронный процесс
initPrimitive	ини элементарный
iniWithTree:	ини с деревом:
inject: into:	ввести: в:
instance	экземпляр
instanceAfter:	экземпляр после:
instanceCount	количество экземпляров
instanceSpecificationOf:	определение экземпляра:
instancesOf:	экземпляры для:
instanceVariableNames:	имена переменных экземпляра:

instanceVariablesArrayIn:	цепь переменных экземпляра на:
instanceVariablesNames:	именаПеременныхЭкземпляра:
instanceVariablesString	цепь переменных экземпляра
instantiateClass: withBytes:	экземпляр класса: с байтами:
instantiateClass: withPointers:	экземпляр класса: с указателями:
instantiateClass: withWords:	экземпляр класса: со словами:
instructionPointerOfContext:	указатель инструкции контекста:
instVarAt:	пер экз от:
instVarAt: put:	пер экз от: пом:
instVarNames	имена переменных экземпляра
integerObjectOf:	объект целое для:
integerPart	целая часть
integerValueOf:	значение целого для:
intern:	содержащий:
internCharacter:	содержащий знак:
interpret	интерпретировать
intersect:	пересечь:
intersects:	пересекает:
isBits	это биты
isBlockContext:	это контекст блока:
isBytes	это байты
isCharacter	это знак
isDigit	это цифра
isDigit: radix:	это цифра: основание:
isEmpty	пустой
isEmptyList:	это пустой список:
isFalse	это ложь
isIndexable:	это нумерованный:
isIntegerObject:	это объект целое:
isIntegerValue:	это значение целого:
isItemizable	детализированный
isKindOf:	это разновидность:
isLastInstance:	это последний экземпляр:
isLeaf	это лист

isLetter	это буква
isLetterSelector:	это селектор из букв:
isLowercase	это в нижнем регистре
isMemberOf:	это член:
isNil	это пусто
isOff	выключен
isOn	включен
isOriginal	это оригинал
isoLanguage:	язык ИСО
isPointers	это указатели
isPointers:	это указатели:
isReceiver	это получатель
isSelf	это сам
isSpecialVariable	это специальная переменная
isSuper	это над
isThisContext	это этот контекст
isTrue	это истина
isUppercase	это в верхнем регистре
isWords	это слова
isWords:	это слова:

J

jump:	прыгнуть:
jumpBytecode	байткод прыжка
jumplf: by:	прыгнуть если: на:

K

key	ключ
key: value:	ключ: значение:
keyAtValue:	ключ от значения:
keyAtValue: ifAbsent:	ключ от значения: если нету:
keys	ключи
keysAndValuesDo:	делать с ключами и значениями:

keysDo:	ключи делать:
keywords	ключевые слова
kindOfSubclass	вид подкласса

L

labelString	цепь бирка
language	язык
language:	язык:
largeContextFlagOf:	флаг большого контекста:
last	последний
lcm:	нок:
left	левый
left:	левый:
left: right:	левый: правый:
length	длина
lengthOf:	длина для:
level:	уровень:
lights:	ламп:
listItem	элемент списка
listItem:	элемент списка:
literal	литерал
literal:	литерал:
literal: ofMethod:	литерал: из метода:
literalCountOf:	количество литералов:
literalCountOfHeader:	количество литералов заголовка:
literals	литералы
ln	лн
load:	загрузить:
location	положение
log	лог
log:	лог:
longUnconditionalJump	длинный безусловный прыжок
lookupMethodInClass:	искать метод в классе:
lookupMethodInDictionary:	искать метод в словаре:

lowByteOf:	младший байт:
lowIOPriority	низкий приоритет ВВ

M

makeRoomAtLast	создать пространство в конце
match:	сопоставить с:
max:	макс:
merge:	объединить:
messageSize	размер сообщения
messages:	сообщения:
method	метод
methodArgument	аргумент метода
methodClassOf:	класс метода:
methodDictionary	словарь методов
methodDictionary:	словарь методов:
methodSelector	селектор метода
millisecondClockValue	значение часов в миллисекундах
millisecondsToRun:	миллисекунд на выполнение:
min:	мин:
minusOne	минус единица
minutes	минуты
multilinePrintArrayOn: ident:	многострочная печать ряда в: отступ:
multilinePrintOn: ident:	многострочная печать в: отступ:
mustBeBoolean	должен быть логическим

N

Number	Число
name	имя
name:	имя:
named:	с именем:

nameOfDay:	имя дня:
nameOfMonth:	имя месяца:
negated	минус
negative	отрицательное
new	новый
new:	новый:
newActiveContext:	новый активный контекст:
newDay: month: year:	новый день: месяц: год:
newDay: year:	новый день: год:
newIndexOf:	новый номер для:
newProcess	новый процесс
newProcessWith:	новый процесс с:
next	следующий
next:	следующий:
next: put:	следующими: пом:
nextInstance	следующий экземпляр
nextLink	следующая связь
nextLink:	следующая связь:
nextMatchFor:	следующий совпадает с:
nextNumber:	следующее число:
nextNumber: put:	следующим числом: пом:
nextPut:	пом следующим:
nextPutAll:	пом следующими все:
nextString	следующая цепь
nextStringPut:	пом следующими цепь:
nextTo:	следующий за:
nextWakeup	следующее пробуждение
nextWord	следующее слово
nextWordPut:	пом следующим словом:
nil	пусто
nilContextFields	очистить поля контекста
ninth	девятый
nodePriority:	приоритет узла:
noMask:	не маска:
not	не
notNil	не пусто
now	текущее

numerator: denominator:	числитель: знаменатель:

O

objectAt:	объект от:
objectAt: put:	объект от: пом:
objectPointerCountOf:	количество указателей объектов для:
occurrencesOf:	вхождений:
odd	нечётное
of: at:	с: за:
on:	на:
on: from: to:	на: от: до:
one	единица
open	открыть
operand:	операнд:
or:	или:
or: or:	или: или:
or: or: or:	или: или: или:
or: or: or: or:	или: или: или: или:
orderedList	упорядоченный список
orderedList:	упорядоченный список:
origin	начало
origin: corner:	начало: угол:
origin: extent:	начало: размеры:
originalFor:	оригинал для:
originalForGlobalVariable:	оригинал для глобальной переменной:
originalForInstanceVariable:	оригинал для переменной экземпляра:
originalForTempVariable:	оригинал для временной переменной:
originalForTempVariable: inMethod:	оригинал для временной переменной: в методе:
originalLanguageFor:	оригинальный язык для:
originalNameForGlobalVariable:	оригинальное имя для глобальной переменной:

originalOrNilFor:	оригинал или пусто для:
originalSymbol	оригинальный символ

P

padTo: put:	заполнить до: пом:
paragraph	параграф
paragraph:	параграф:
parent:	родитель:
parentDictionary:	родительский словарь:
parse:	анализ:
pc	ск
pc:	ск:
peek	считать
peekFor:	считать для:
perform:	выполнить:
perform: with:	выполнить: с:
perform: with: with:	выполнить: с: с:
perform: with: with: with:	выполнить: с: с: с:
perform: withArguments:	выполнить: с аргументами:
pi	пи
poolDictionaries:	словари пула:
pop:	вытолкнуть:
popInteger	вытолкнуть целое
popStack	вытолкнуть стэк
popStackBytecode	байткод вытолкнуть стэк
position	позиция
positive	положительное
positive16BitIntegerFor:	положительное 16 битное целое для:
positive16BitValueOf:	положительное 16 битное значение для:
pragmas:	прагмы:
precedence	старшинство
precedence:	старшинство:
primitiveAdd	элементарный добавить

primitiveAsFIoat	элементарный как плавающее
primitiveAsObject	элементарный как объект
primitiveAsOop	элементарный как УО
primitiveAt	элементарный от
primitiveAtEnd	элементарный в конце
primitiveAtPut	элементарный от пом
primitiveBeCursor	элементарный стать курсором
primitiveBecome	элементарный становится
primitiveBeDisplay	элементарный стать экраном
primitiveBitAnd	элементарный побитовое и
primitiveBitOr	элементарный побитовое или
primitiveBitShift	элементарный сдвинуть биты
primitiveBitXor	элементарный побитовое исключающее или
primitiveBlockCopy	элементарный экземпляр блока
primitiveClass	элементарный метод класс
primitiveCopyBits	элементарный копировать биты
primitiveCoreLeft	элементарный оставшаяся память
primitiveCursorLink	элементарный привязать курсор
primitiveCursorLocPut	элементарный поместить курсор в положение
primitiveDiv	элементарный разд
primitiveDivide	элементарный разделить
primitiveDrawLoop	элементарный цикл отрисовки
primitiveEqual	элементарный равно
primitiveEquivalent	элементарный эквивалентен
primitiveExitToDebugger	элементарный выйти в отладчик
primitiveExponent	элементарный экспонента
primitiveFail	неудача элементарного метода
primitiveFIoatAdd	элементарный добавить плавающее

primitiveFloatDivide	элементарный плавающее разделить
primitiveFloatEqual	элементарный плавающее равно
primitiveFloatGreaterOrEqual	элементарный плавающее больше или равно
primitiveFloatLessOrEqual	элементарный плавающее меньше или равно
primitiveFloatLessThan	элементарный плавающее меньше чем
primitiveFloatMultiply	элементарный плавающее умножить
primitiveFloatNotEqual	элементарный плавающее не равно
primitiveFloatSubtract	элементарный вычесть плавающее
primitiveFloatGreaterThan	элементарный плавающее больше чем
primitiveFlushCache	элементарный очистить кэш
primitiveFractionalPart	элементарный дробная часть
primitiveGreaterOrEqual	элементарный больше или равно
primitiveGreaterThan	элементарный больше чем
primitiveIndexOf:	номер элементарного метода:
primitiveInputSemaphore	элементарный семафор ввода
primitiveInputWord	элементарный ввести слово
primitiveInstVarAt	элементарный пер экз от
primitiveInstVarAtPut	элементарный пер экз от пом
primitiveLessOrEqual	элементарный меньше или равно
primitiveLessThan	элементарный меньше чем
primitiveMakePoint	элементарный создать точку
primitiveMod	элементарный деление по модулю
primitiveMousePoint	элементарный точка мыши
primitiveMultiply	элементарный умножить
primitiveNew	элементарный новый

primitiveNewMethod	элементарный новый метод
primitiveNewWithArg	элементарный новый с аргументом
primitiveNext	элементарный следующий
primitiveNextInstance	элементарный следующий экземпляр
primitiveNextPut	элементарный пом следующим
primitiveNotEqual	элементарный не равно
primitiveObjectAt	элементарный Объект от
primitiveObjectAtPut	элементарный Объект от пом
primitiveOopsLeft	элементарный оставшиеся УО
primitivePerform	элементарный выполнить
primitivePerformWithArgs	элементарный выполнить с аргументами
primitiveQuit	элементарный выйти
primitiveQuo	элементарный частное
primitiveResponse	ответ элементарного метода
primitiveResume	элементарный возобновить
primitiveSampleInterval	элементарный интервал семафора
primitiveScanCharacters	элементарный просмотреть знаки
primitiveSignal	элементарный сигнал
primitiveSignalAtOopsLeftWordsLeft	элементарный сигналить при оставшихся УО при оставшихся словах
primitiveSignalAtTick	элементарный сигналить на тике
primitiveSize	элементарный размер
primitiveSnapshot	элементарный сделать снимок
primitiveSomeInstance	элементарный некоторый экземпляр
primitiveStringAt	элементарный Цепь от
primitiveStringAtPut	элементарный Цепь от пом
primitiveStringReplace	элементарный заменить цепь
primitiveSubtract	элементарный вычесть
primitiveSuspend	элементарный приостановить

primitiveTickWordsInto	элементарный слова тиков в
primitiveTimesTwoPower	элементарный умножить на два в степени
primitiveTimeWordsInto	элементарный слова времени в
primitiveTruncated	элементарный плавающее усечь
primitiveValue	элементарный значение
primitiveValueWithArgs	элементарный значение с аргументами
primitiveWait	элементарный ждать
primSignal: atMilliseconds:	прим сигнал: при миллисекундах:
printArrayOn:	печатать ряд в:
printCategoryOn: inLanguage:	печатать категорию в: на языке:
printClassVariableNamesOn: language:	печатать имена переменных класса в: на языке:
printClassVariableNamesOn: selector: language:	печатать имена переменных класса в: селектор: язык:
printCommentOn:	печатать комментарий в:
printElementsOn:	печатать элементы в:
printInstanceVariableNamesOn: language:	печатать имена переменных экземпляра в: на языке:
printInstanceVariableNamesOn: selector: language:	печатать имена переменных экземпляра в: селектор: язык:
printNameOn:	печатать имя в:
printNumberOn:	печатать число в:
printOn:	печатать в:
printOn: headerWithArguments:	печатать в: заголовок с аргументами:
printOn: newLineWithIdent:	печатать в: новую строку с отступом:
printOn: statements:	печатать в: предложения:
printReceiverOn: ident: inOneLine:	печатать получателя в: отступ: в одну строку:

printSelectorAndArgumentsOn: ident: inOneLine:	печатать селектор и аргументы в: отступ: в одну строку:
printSharedPoolsOn: inLanguage:	печатать разделяемые пулы в: на языке:
printString	цепь для печати
printTemporariesOn:	печатать временные в:
priority	приоритет
priority:	приоритет:
push:	протолкнуть:
pushActiveContextBytecode	батйкод поместить активный контекст
pushConstantBytecode	байткод поместить константу
pushInteger:	протолкнуть целое:
pushLiteralConstant:	поместить литерал константу:
pushLiteralConstantBytecode	байткод поместить литерал константу
pushLiteralVariable:	поместить литерал переменную:
pushLiteralVariableBytecode	байткод поместить литерал переменную
pushReceiverBytecode	байткод поместить получателя
pushReceiverVariable:	поместить переменную получателя:
pushReceiverVariableBytecode	байткод поместить переменную получателя
pushTemporaryVariable:	поместить временную переменную:
pushTemporaryVariableBytecode	байткод поместить временную переменную

Q

quickInstanceLoad	быстрая загрузка переменной экземпляра
quickReturnSelf	быстрый возврат себя
quo:	частное:

R

r: g: b:	к: з: с:
radiansToDegrees	радианы в градусы
radix	основание
radix:	основание:
raisedTo:	в степени:
raisedToInteger:	в целой степени:
readFrom:	читать из:
readOnlyFileNamed:	файл только для чтения с именем:
receive: from:	получить: из:
receiver	получатель
receiver:	получатель:
receiver: messages:	получатель: сообщения:
receiver: selector: arguments:	получатель: селектор: аргументы:
receiverPrecedence	приоритет получателя
receiverType	тип получателя
reciprocal	обратное
recompile:	перекомпилировать:
reject:	отбросить:
release	освободить
remove:	удалить:
remove: ifAbsent:	удалить: если нету:
removeAll:	удалить все:
removeCategory:	удалить категорию:
removeClassVarName:	удалить имя переменной класса:
removeDependent:	удалить зависимость:
removeFirst	удалить первый
removeFirstLinkOfList:	удалить первую связь списка:
removeFromSystem	удалить из системы
removeIndex:	удалить номер:
removeInstVarName:	удалить имя переменной экземпляра:
removeKey:	удалить ключ:

removeKey: ifAbsent:	удалить ключ: если нету:
removeLast	удалить последний
removeSelector:	удалить селектор:
removeSharedPool:	удалить разделяемый пул:
removeSubclass:	удалить подкласс:
rename:	переименовать:
renderContentOn:	нарисовать содержимое на:
replaceFrom: to: with:	заменить от: до: на:
replaceFrom: to: with: startingAt:	заменить от: до: на: начиная с:
replaceSpecialVariable	заменить специальную переменную
replaceVariable:	заменить переменную:
reset	сбросить
respondsTo:	отвечает на:
rest	остаток
result	результат
result1	результат 1
result2	результат 2
result3	результат 3
resume	возобновить
resume:	возобновить:
resumptionTime	время возобновления
return:	вернуть:
returnAIICardsTo:	вернуть все карты в:
returnBytecode	байткод возврата
returnToActiveContext:	вернуться в активный контекст:
returnValue: to:	вернуть значение: к:
reverseDo:	реверсивно делать:
right	правый
right:	правый:
rightCenter	правый центр
rounded	округлить
roundTo:	округлить до:

S

String	Цепь
Symbol	Символ
sameAs:	такой же как:
scanFor:	просмотреть для:
schedulerPointer	указатель на Планировщика
scopeHas: ifTrue:	в области видимости есть: истина:
second	второй
select:	выбрать:
selectedClass	выбранный класс
selector	селектор
selector:	селектор:
selector: block: pragmas: comment:	селектор: блок: прагмы: комментарий:
selector: block: pragmas: comment: tail:	селектор: блок: прагмы: комментарий: хвост:
selector: language:	селектор: язык:
selectorAsSymbol	селектор как символ
selectors	селекторы
selectorsDictionary	словарь селекторов
selectSubclasses:	выбрать подклассы:
selectSuperclasses:	выбрать надклассы:
self	сам
sendBytecode	байткод посылки
sender	отправитель
sendLiteralSelectorBytecode	байткод послать селектор литерал
sendMustBeBoolean	послать должен быть логическим
sendSelector: argumentCount:	послать селектор: количество аргументов:
sendSelectorToClass:	послать селектор классу:
sendSpecialSelectorBytecode	байткод послать специальный селектор
separator	разделитель

setCards	присвоить карты
setCollection:	присвоить набор:
setCollection: map:	присвоить набор: карту:
setContents:	присвоить содержимое:
setData:	присвоить данные:
setElement:	присвоить элементу:
setFrom: to: by:	присвоить от: до: через:
setOff	выключенный
setOn	включенный
setOn:	установить на:
setToEnd	установить в конец
setVariables	присвоить переменные
seventh	седьмой
shade:	тень:
shallowCopy	поверхностная копия
sharedPools	разделяемые пулы
sharedPoolsString	цепь разделяемых пулов
shortConditionalJump	короткий условный прыжок
shortUnconditionalJump	короткий безусловный прыжок
shouldNotImplement	не должен реализовывать
show:	показать:
shuffle	тасовать
sign	знак
signal	сигнал
simpleReturnValue: to:	просто вернуть значение: к:
simplify	упростить
sin	син
singleExtendedSendBytecode	байткод посылки с одним расширением
singleExtendedSuperBytecode	байткод посылки наду с одним расширением
sixth	шестой
size	размер
sizeFor:	размер для:
skip:	пропустить:
skipTo:	пропустить до:
sleep:	сон:

someInstance	некоторый экземпляр
sort	сортированный
sortBlock:	сортирующий блок:
sourceCodeAt:	исходный текст от:
sourceFor: inLanguage:	исходный текст для: на языке:
sourceMethodAt:	исходный метод от:
sources	исходники
sourceText	исходный текст
sourceTextIn: class:	исходный текст на: класс:
sourceTextIn: with: class:	исходный текст на: с: класс:
space	пробел
span	промежуток
specialSelectorPrimitive-Response	ответ элементарного метода специального селектора
species	разновидность
spend: for:	потратить: на:
spend: for: deducting:	потратить: на: вычесть:
spendDeductible: for:	потратить на налоги: на:
sqrt	квадратный корень
squared	в квадрате
stackBytecode	байткод стэка
stackPointerOfContext:	указатель стэка контекста:
stackTop	вершина стэка
stackValue:	значение стэка:
statements	предложения
statements:	предложения:
store:	поместить:
storeAndPopReceiverVariable-Bytecode	байткод сохранить в переменную экземпляра и вытолкнуть
storeAndPopTemporary-VariableBytecode	байткод сохранить во временную переменную и вытолкнуть
storeByte: ofObject: withValue:	сохранить байт: в объект: со значением:
storeContextRegisters	сохранить регистры контекста

storeInstructionPointerValue: inContext:	сохранить значение указателя инструкции: в контекст:
storeInteger: ofObject: withValue:	сохранить целое: в объект: со значением:
storeOn:	поместить в:
storePointer: ofObject: withValue:	сохранить указатель: в объект: со значением:
storeStackPointerValue: inContext:	сохранить значение указателя стэка: в контекст:
storeString	цепь для помещения
storeWord: ofObject: withValue:	сохранить слово: в объект: со значением:
streamContents:	содержание потока:
strictlyPositive	строго положительное
string	цепь
style:	стиль:
subclass:	подкласс:
subclass: instanceVariableNames: classVariableNames: poolDictionaries: category:	подкласс: имена переменных экземпляра: имена переменных класса: словари пула: категория:
subclass: uses: instanceVariableNames: classVariableNames: poolDictionaries: category:	подкласс: используются: имена переменных экземпляра: имена переменных класса: словари пула: категория:
subclasses	подклассы
subclassInstVarNames	имена переменных экземпляра подклассов
subclassResponsibility	ответственность подкласса
subscript: with:	подномер: с:
subscript: with: storing:	подномер: с: сохранить:
subtractDate:	вычесть дату:
subtractDays:	вычесть дни:
subtractTime:	вычесть время:
success	успех
success:	успех:
superAt: put:	над од: пом:

superclass	надкласс
superclass:	надкласс:
superclassOf:	надкласс для:
superSend	послать наду
superSend:	послать наду:
suspend	приостановить
suspendActive	приостановить активный
swapPointersOf: and:	обменять указатель: и:
switchAndInstallFontToID:	переключиться и установить шрифт для ИД:
symbolsDictionary	символы
synchronousSignal:	синхронный сигнал:
systemBackgroundPriority	приоритет фона системы

T

tab	таб
table:	таблица:
tableRow	строка таблицы
tableRow:	строка таблицы:
tag:	тэг:
tail:	хвост:
take:	взять:
tan	тан
target:	цель:
target: type:	цель: тип:
targetPC	счётчик команд цели
templateForTranslateSymbolsForMethod: in:	шаблон перевода символов для метода: в:
templateForTranslateSymbolsIn: dictionary:	шаблон для перевода символов на: словарь:
temporaries	временные
temporaries:	временные:
temporariesNumber	количество временных
temporary	временный
temporary:	временная:

temporaryCountOf:	количество временных:
temporaryNames	имена временных
temporaryVariable	временная переменная
terminate	завершить
terminateActive	завершить активный
test	тест
text:	текст:
thereIsTranslationFor:	существует перевод для:
thereIsTranslationFor: into:	существует перевод для: на:
thinSpace	узкий пробел
third	третий
timesRepeat:	раз повторить:
timesTwoPower:	умножить на два в степени:
timingPriority	приоритет синхронных процессов
to:	до:
to: by:	до: через:
to: by: do:	до: через: делать:
to: do:	до: делать:
to: index:	в: номер:
to: selector: index: argumentsCount:	к: селектор: номер: количество аргументов:
today	сегодня
top	вершина
totalDeductions	всего налогов
totalReceivedFrom:	общее поступление из:
totalSeconds	всего секунд
totalSpentFor:	общие траты на:
transfer: fromField: ofObject: toField: ofObject:	перенести: от номера: из объекта: в номер: в объект:
transferTo:	перейти к:
translate: from: to: as:	перевести: с: на: как:
translateClassVariablesAs:	перевести переменные класса как:
translated	переведённый
translateIn: selectorPart:	перевести на: часть селектора:
translateIn: with:	перевести на: с:

translateIn: with: class:	перевести на: с: класс:
translateInstanceVariable-NamesIn:	перевести имена переменных экземпляра на:
translateInstanceVariablesAs:	перевести переменные экземпляра как:
translateMethodVariables	перевести переменные метода
translateNumbersIn:	перевести числа на:
translatePartUsesIn:	перевести часть используются на:
translateSelector	перевести селектор
translateSelectorIn:	перевести селектор на:
translateSelectorsIn:	перевести селекторы на:
translateSymbols	перевести символы
translateSymbolsAs:	перевести символы как:
translateSymbolsIn:	перевести символы на:
translateVariablesForMethod: as:	перевести переменные метода: как:
translateVariablesIn: with: class:	перевести переменные на: с: класс:
translationFor: into:	перевод для: на:
translationForSelector: into:	перевод для селектора: на:
translationForSpecialVariable: into:	перевод специальной переменной: на:
tree:	дерево:
tree: parentDictionary:	дерево: родительский словарь:
true	истина
truncated	усечь
truncateTo:	усечь до:
turnOff	выключить
turnOn	включить
turnOn:	включить:
two	двойка
type:	тип:
type: index:	тип: номер:
typeAndIndexOfVariable:	тип и номер переменной:

U

unorderedList	неупорядоченный список
unorderedList:	неупорядоченный список:
unPop:	отменить выталкивание:
update:	обновить:
upTo:	вплоть до:
userBackgroundPriority	приоритет фона пользователя
userInterruptPriority	приоритет прерываний пользователя
userSchedulingPriority	приоритет интерфейса с пользователем

V

value	значение
value:	значение:
value: type:	значение: тип:
value: value:	значение: значение:
value: value: value:	значение: значение: значение:
values	значения
variable	переменная
variable:	переменная:
variable: value:	переменная: значение:
variableByteSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:	переменный подкласс байтов: имена переменных экземпляра: имена переменных класса: словари пула: категория:
variableSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:	переменный подкласс: имена переменных экземпляра: имена переменных класса: словари пула: категория:
variableWordSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:	переменный подкласс слов: имена переменных экземпляра: имена переменных класса: словари пула: категория:

verify:	проверить:
verifyIn:	проверить в:
verifyNumber	проверить число

W

wait	ждать
waitForWakeup	ждать пробуждения
wakeHighestPriority	разбудить процесс с наивысшим приоритетом
wakeup	проснуться
whichCategoryIncludesSelector:	какая категория содержит селектор:
whichClassIncludesSelector:	какой класс содержит селектор:
whichSelectorsAccess:	которые селекторы обращаются к:
whichSelectorsReferTo:	которые селекторы ссылаются на:
whileFalse	пока ложь
whileFalse:	пока ложь:
whileTrue	пока истина
whileTrue:	пока истина:
width	ширина
with:	с:
with: do:	с: делать:
with: from: to:	с: от: до:
with: with:	с: с:
with: with: with:	с: с: с:
with: with: with: with:	с: с: с: с:
with: with: with: with: with:	с: с: с: с: с:
with: with: with: with: with: with:	с: с: с: с: с: с:
withAll:	со всеми:
withAllSubclasses	со всеми подклассами
withoutTail	без хвоста

withStyleFor: do:	со стилем для: делать:
withTail	с хвостом
word	слово
word:	слово:

X

x	икс
x: y:	икс: игрек:
xor:	и или:

Y

y	игрек
yield	уступить
yourself	себя

Z

zero	ноль

32.2 Русско-английский словарь селекторов

A

активировать новый метод	activateNewMethod
активный приоритет	activePriority
активный процесс	activeProcess
анализ:	parse:
аргумент блока	blockArgument
аргумент метода	methodArgument
аргументы	arguments

аргументы:	arguments:
аргументы: временные: предложения:	arguments: temporaries: statements:
асинхронный сигнал:	asynchronousSignal:
ассоциации делать:	associationsDo:
ассоциация для переменной класса: в классе:	associationForClassVariable: inClass:
ассоциация от:	associationAt:
ассоциация от: если нету:	associationAt: ifAbsent:
атрибут от:	attributeAt:
атрибут от: пом:	attributeAt: put:
атрибуты	attributes
атрибуты:	attributes:

Б

байткод возврата	returnBytecode
байткод вытолкнуть стэк	popStackBytecode
байткод поместить временную переменную	pushTemporaryVariableBytecode
байткод поместить константу	pushConstantBytecode
байткод поместить литерал константу	pushLiteralConstantBytecode
байткод поместить литерал переменную	pushLiteralVariableBytecode
байткод поместить переменную получателя	pushReceiverVariableBytecode
байткод поместить получателя	pushReceiverBytecode
байткод послать селектор литерал	sendLiteralSelectorBytecode
байткод послать специальный селектор	sendSpecialSelectorBytecode
байткод посылки	sendBytecode
байткод посылки наду с двумя расширениями	doubleExtendedSuperBytecode
байткод посылки наду с одним расширением	singleExtendedSuperBytecode

байткод посылки с двумя расширениями	doubleExtendedSendBytecode
байткод посылки с одним расширением	singleExtendedSendBytecode
байткод прыжка	jumpBytecode
байткод расширенное помещение	extendedPushBytecode
байткод сохранить в переменную экземпляра и вытолкнуть	storeAndPopReceiverVariableBytecode
байткод сохранить во временную переменную и вытолкнуть	storeAndPopTemporaryVariableBytecode
байткод стэка	stackBytecode
байткод удвоить вершину	duplicateTopBytecode
байткоды для:	bytecodesFor:
батйкод поместить активный контекст	pushActiveContextBytecode
без хвоста	withoutTail
блок	block
блок:	block:
блок цитата	blockquote
блок цитата:	blockquote:
быстрая загрузка переменной экземпляра	quickInstanceLoad
быстрый возврат себя	quickReturnSelf

В

в: номер:	to: index:
в верхнем регистре	asUppercase
в квадрате	squared
в конце	atEnd
в нижнем регистре	asLowercase
в области видимости есть: истина:	scopeHas: ifTrue:

в степени:	raisedTo:
в целой степени:	raisedToInteger:
ввести: в:	inject: into:
вернуть:	return:
вернуть все карты в:	returnAIICardsTo:
вернуть значение: к:	returnValue: to:
вернуться в активный контекст:	returnToActiveContext:
вершина	top
вершина стэка	stackTop
взять:	take:
вид подкласса	kindOfSubclass
включен	isOn
включенный	setOn
включить	turnOn
включить:	turnOn:
возобновить	resume
возобновить:	resume:
восьмой	eighth
вплоть до:	upTo:
временная:	temporary:
временная переменная	temporaryVariable
временные	temporaries
временные:	temporaries:
временный	temporary
время возобновления	resumptionTime
все имена переменных класса	allClassVarNames
все имена переменных экземпляра	allInstVarNames
все надклассы	allSuperclasses
все подклассы	allSubclasses
все разделяемые пулы	allSharedPools
все селекторы	allSelectors
все экземпляры	allInstances
всего налогов	totalDeductions
всего секунд	totalSeconds
вся маска:	allMask:

второй	second
вхождений:	occurrencesOf:
выбранный класс	selectedClass
выбрать:	select:
выбрать надклассы:	selectSuperclasses:
выбрать подклассы:	selectSubclasses:
вывести байткоды в:	emitBytecodesOn:
вывести без последнего в: с:	emitExceptLastOn: with:
вывести в файл	fileOut
вывести в файл:	fileOutOn:
вывести в файл изменённые сообщения: в:	fileOutChangedMessages: on:
вывести в файл категорию:	fileOutCategory:
вывести код для выполнения в: с:	emitForEvaluatedValueOn: with:
вывести код для запоминания в: с:	emitStoreOn: with:
вывести код для запоминания и извлечения в: с:	emitStorePopOn: with:
вывести код для значения в: с:	emitForValueOn: with:
вывести код для значения в: с: посылать к:	emitForValueOn: with: sendTo:
вывести код для эффекта в: с:	emitForEffectOn: with:
вывести код для эффекта в: с: посылать к:	emitForEffectOn: with: sendTo:
вызвавший	caller
выключен	isOff
выключенный	setOff
выключить	turnOff
выполнить:	perform:
выполнить: с:	perform: with:
выполнить: с: с:	perform: with: with:
выполнить: с: с: с:	perform: with: with: with:
выполнить: с аргументами:	perform: withArguments:
выполнить арифметический элементарный метод	dispatchArithmeticPrimitives
выполнить новый метод	executeNewMethod

выполнить собственный элементарный метод	dispatchPrivatePrimitives
выполнить управляющий элементарный метод	dispatchControtPrimitives
выполнить элементарные метоы нумерации и Потоков	dispatchSubscriptAndStream-Primitives
выполнить элементарный метод	dispatchPrimitives
выполнить элементарный метод большого целого	dispatchLargeIntegerPrimitives
выполнить элементарный метод ввода вывода	dispatchlnputOutputPrimitives
выполнить элементарный метод плавающего	dispatchFIoatPrimitives
выполнить элементарный метод системы	dispatchSystemPrimitives
выполнить элементарный метод управления хранилищем	dispatchStorageManagement-Primitives
выполнить элементарный метод целого	dispatchIntegerPrimitives
выполнить этот байткод	dispatchOnThisBytecode
выражение:	expression:
выражение: уровень:	expression: level:
высокий приоритет ВВ	highIOPriority
высота	height
вытолкнуть:	pop:
вытолкнуть стэк	popStack
вытолкнуть целое	popInteger
вычесть время:	subtractTime:
вычесть дату:	subtractDate:
вычесть дни:	subtractDays:
выявить:	detect:
выявить: если ни одного:	detect: ifNone:

Г

гарантировать:	ensure:

глубокая копия	deepCopy
горизонтальная линия	horizontalRule
градусы в радианы	degreesToRadians

Д

данные:	data:
данные определения	definitionData
данные определения:	definitionData:
двойка	two
девятый	ninth
декомпилировать:	decompile:
делать:	do:
делать: с разделителем:	do: separatedBy:
делать для все подэкземпляров:	allSubInstancesDo:
делать для всех надклассов:	allSuperclassesDo:
делать для всех подклассов:	allSubclassesDo:
делать для всех экземпляров:	allInstancesDo:
делать с ключами и значениями:	keysAndValuesDo:
день недели:	dayOfWeek:
дерево:	tree:
дерево: родительский словарь:	tree: parentDictionary:
детализированный	isItemizable
длина	length
длина для:	lengthOf:
длина цифр	digitLength
длинный безусловный прыжок	longUnconditionalJump
длинный условный прыжок	IongConditionalJump
для:	for:
для взаимного исключения	forMutualExclusion
для всех переменных метода делать:	forAllMethodVariablesDo:
для всех символов делать:	forAllSymbolsDo:
дней в году:	daysInYear:

дней в месяце: для года:	daysInMonth: forYear:
до:	to:
до: делать:	to: do:
до: через:	to: by:
до: через: делать:	to: by: do:
добавить:	add:
добавить: перед:	add: before:
добавить: после:	add: after:
добавить: с вхождениями:	add: withOccurrences:
добавить аргументы:	addArguments:
добавить аргументы: и временные переменные:	addArguments: andTemporaryVariables:
добавить ассоциацию для глобальной переменной:	addAssociationForGlobalVariable:
добавить ассоциацию для глобальной переменной: в классе:	addAssociationForGlobalVariable: inClass:
добавить ассоциацию для переменной окружения: в классе:	addAssociationForEnvironmentVariable: inClass:
добавить временную переменную:	addTemporaryVariable:
добавить временные переменные:	addTemporaryVariables:
добавить время:	addTime:
добавить все:	addAll:
добавить дни:	addDays:
добавить зависимость:	addDependent:
добавить имя переменной класса:	addClassVarName:
добавить имя переменной экземпляра:	addInstVarName:
добавить литерал:	addLiteral:
добавить литерал: узел:	addLiteral: node:
добавить литералы к:	addLiteralsTo:
добавить первым:	addFirst:
добавить первыми все:	addAllFirst:

добавить последней связью: к списку:	addLastLink: toList:
добавить последним:	addLast:
добавить последними все:	addAllLast:
добавить разделяемый пул:	addSharedPool:
добавить селектор: с методом:	addSelector: withMethod:
добавить стиль:	addStyle:
документ	document
документ:	document:
должен быть логическим	mustBeBoolean
достать целое: из объекта:	fetchInteger: ofObject:
дробная часть	fractionPart

Е

единица	one
если не пустой:	ifNotEmpty:
если не пустой: если пустой:	ifNotEmpty: ifEmpty:
если пустой:	ifEmpty:
если пустой: если не пустой:	ifEmpty: ifNotEmpty:

Ж

ждать	wait
ждать пробуждения	waitForWakeup

З

Знак	Character
завершить	terminate
завершить активный	terminateActive
зависимости	dependents
заголовок	heading
заголовок:	heading:

заголовок браузера по умолчанию	defaultBrowserTitle
загрузить:	load:
заканчивается на:	endsWith:
закрыть	close
заменить от: до: на:	replaceFrom: to: with:
заменить от: до: на: начиная с:	replaceFrom: to: with: startingAt:
заменить переменную:	replaceVariable:
заменить специальную переменную	replaceSpecialVariable
заполнить до: пом:	padTo: put:
знак	sign
значение	value
значение:	value:
значение: значение:	value: value:
значение: значение: значение:	value: value: value:
значение: тип:	value: type:
значение АСКОИ	asciiValue
значение стэка:	stackValue:
значение флага:	flagValueOf:
значение целого для:	integerValueOf:
значение часов в миллисекундах	millisecondClockValue
значения	values

И

и:	and:
и: и:	and: and:
и или:	xor:
игрек	y
из:	from:
из: значение:	from: value:
из дней:	fromDays:
из секунд:	fromSeconds:

из цепи:	fromString:
извлечь байт	fetchByte
извлечь байт: из объекта:	fetchByte: ofObject:
извлечь биты от: до: из:	extractBits: to: of:
извлечь длину в байтах:	fetchByteLengthOf:
извлечь длину в словах:	fetchWordLengthOf:
извлечь класс:	fetchClassOf:
извлечь регистры контекста	fetchContextRegisters
извлечь слово: из объекта:	fetchWord: ofObject:
извлечь указатель: из объекта:	fetchPointer: ofObject:
изменён	changed
изменить перевод для: на:	changeTranslationFor: on:
изменить предел страховки:	changeInsurenceLimit:
изображение	image
изображение:	image:
икс	x
икс: игрек:	x: y:
или:	or:
или: или:	or: or:
или: или: или:	or: or: or:
или: или: или: или:	or: or: or: or:
имеет методы	hasMethods
имена аргументов	argumentNames
имена временных	temporaryNames
имена переменных класса	classVarNames
имена переменных класса:	classVariableNames:
имена переменных экземпляра	instVarNames
имена переменных экземпляра:	instanceVariableNames:
имена переменных экземпляра подклассов	subclassInstVarNames
именаПеременныхЭкземпляра:	instanceVariablesNames:
имя	name
имя:	name:
имя дня:	nameOfDay:
имя месяца:	nameOfMonth:
имя файла:	fileNamed:

ини с деревом:	iniWithTree:
ини элементарный	initPrimitive
инициализировать	initialize
инициализировать:	initialize:
инициализировать баланс:	initialBalance:
инициализировать кэш методов	initializeMethodCache
инициализировать налоги	initializeDeductions
инициализировать номер Ассоциации	initializeAssociationIndex
инициализировать номера Знака	initializeCharacterIndex
инициализировать номера класса	initializeClassIndices
инициализировать номера контекста	initializeContextlndices
инициализировать номера Планировщика	initializeSchedulerIndices
инициализировать номера Потока	initializeStreamIndices
инициализировать номера Сообщения	initializeMessageIndices
инициализировать номера Точки	initializePointIndices
инициализировать синхронный процесс	initializeTimingProcess
инициализировать словарь оснований	initializeRadixDictionary
инициализировать словарь экспонент	initializeExponentsDictionary
инициализировать специальные имена	initializeSpecialNames
интерпретировать	interpret
искать метод в классе:	lookupMethodInClass:
искать метод в словаре:	lookupMethodInDictionary:
истина	true
истина:	ifTrue:

истина: ложь:	ifTrue: ifFalse:
исходники	sources
исходный метод от:	sourceMethodAt:
исходный текст	sourceText
исходный текст для: на языке:	sourceFor: inLanguage:
исходный текст на: класс:	sourceTextIn: class:
исходный текст на: с: класс:	sourceTextIn: with: class:
исходный текст от:	sourceCodeAt:

К

к: з: с:	r: g: b:
к: селектор: номер: количество аргументов:	to: selector: index: argumentsCount:
как:	as:
как дробь	asFraction
как знак	asCharacter
как мешок	asBag
как множество	asSet
как объект	asObject
как оригинальный литерал	asOriginalLiteral
как плавающее	asFloat
как предложение	asStatement
как ряд	asArray
как ряд байтов	asByteArray
как секунды	asSeconds
как символ	asSymbol
как сортированный набор	asSortedCollection
как сортированный набор:	asSortedCollection:
как упорядоченный набор	asOrderedCollection
как целое	asInteger
как цепь	asString
какая категория содержит селектор:	whichCategoryIncludesSelector:
какой класс содержит селектор:	whichClassIncludesSelector:

категория	category
категория:	category:
квадратный корень	sqrt
кисть:	brush:
класс	class
класс:	class:
класс: если:	class: if:
класс метода:	methodClassOf:
класс содержимого	contentsClass
классы	classes
ключ	key
ключ: значение:	key: value:
ключ от значения:	keyAtValue:
ключ от значения: если нету:	keyAtValue: ifAbsent:
ключевые слова	keywords
ключи	keys
ключи делать:	keysDo:
код	code
код:	code:
количество:	count:
количество аргументов	argumentsNumber
количество аргументов:	argumentCountOf:
количество аргументов блока:	argumentCountOfBlock:
количество врем пер	amountTempVars
количество временных	temporariesNumber
количество временных:	temporaryCountOf:
количество литералов:	literalCountOf:
количество литералов заголовка:	literalCountOfHeader:
количество наличных	cashOnHand
количество указателей объектов для:	objectPointerCountOf:
количество экземпляров	instanceCount
комментарий	comment
комментарий:	comment:
компилировать:	compile:

компилировать: классифицировать:	compile: classified:
компилировать: классифицировать: уведомлять:	compile: classified: notifying:
компилировать: уведомлять:	compile: notifying:
компилировать весь	compileAll
копировать: из:	copy: from:
копировать: из: классифицировать:	copy: from: classified:
копировать все: из:	copyAll: from:
копировать все: из: классифицировать:	copyAll: from: classified:
копировать все категории из:	copyAllCategoriesFrom:
копировать категорию: из:	copyCategory: from:
копировать категорию: из: классифицировать:	copyCategory: from: classified:
копия	copy
копия без:	copyWithout:
копия от: до:	copyFrom: to:
копия с:	copyWith:
копия с заменой всех: на:	copyReplaceAll: with:
копия с заменой от: до: на:	copyReplaceFrom: to: with:
короткий безусловный прыжок	shortUnconditionalJump
короткий условный прыжок	shortConditionalJump
которые селекторы обращаются к:	whichSelectorsAccess:
которые селекторы ссылаются на:	whichSelectorsReferTo:
критический:	critical:

Л

ламп:	lights:
левый	left
левый:	left:
левый: правый:	left: right:

литерал	literal
литерал:	literal:
литерал: из метода:	literal: ofMethod:
литералы	literals
лн	ln
лог	log
лог:	log:
ложь	false
ложь:	ifFalse:
ложь: истина:	ifFalse: ifTrue:
любой из маски:	anyMask:
любой удовлетворяет:	anySatisfy:

М

макс:	max:
между: и:	between: and:
метод	method
миллисекунд на выполнение:	millisecondsToRun:
мин:	min:
минус	negated
минус единица	minusOne
минуты	minutes
младший байт:	lowByteOf:
многострочная печать в: отступ:	multilinePrintOn: ident:
многострочная печать ряда в: отступ:	multilinePrintArrayOn: ident:
модуль	abs
может понимать:	canUnderstand:

Н

на:	on:
на: от: до:	on: from: to:

на миллисекунды:	forMilliseconds:
на секунды:	forSeconds:
набор: карта:	collection: map:
над од: пом:	superAt: put:
надкласс	superclass
надкласс:	superclass:
надкласс для:	superclassOf:
найти новый метод в классе:	findNewMethodInClass:
найти первый:	findFirst:
найти последний:	findLast:
найти элемент или пусто:	findElementOrNil:
нарисовать содержимое на:	renderContentOn:
наследует от:	inheritsFrom:
настроить временные	fixTemps
начало	origin
начало: размеры:	origin: extent:
начало: угол:	origin: corner:
начальный указатель инструкции метода:	initialInstructionPointerOfMethod:
начальный экземпляр для:	initialInstanceOf:
начинается с:	beginsWith:
не	not
не должен реализовывать	shouldNotImplement
не маска:	noMask:
не понимаю:	doesNotUnderstand:
не пусто	notNil
не пусто:	ifNotNil:
не пусто: пусто:	ifNotNil: ifNil:
невозможно вернуть:	cannotReturn:
некоторый экземпляр	someInstance
неудача элементарного метода	primitiveFail
неупорядоченный список	unorderedList
неупорядоченный список:	unorderedList:
нечётное	odd
низкий приоритет ВВ	lowIOPriority
новый	new
новый:	new:

новый активный контекст:	newActiveContext:
новый день: год:	newDay: year:
новый день: месяц: год:	newDay: month: year:
новый номер для:	newIndexOf:
новый процесс	newProcess
новый процесс с:	newProcessWith:
нод:	gcd:
нок:	lcm:
ноль	zero
номер:	index:
номер глобальной переменной:	indexOfGlobalVariable:
номер для:	indexOf:
номер для: если нету:	indexOf: ifAbsent:
номер литерала:	indexOfLiteral:
номер месяца:	indexOfMonth:
номер основания	indexOfRadix
номер поднабора: начиная с:	indexOfSubCollection: startingAt:
номер поднабора: начиная с: если нету:	indexOfSubCollection: startingAt: ifAbsent:
номер поля для:	fieldIndexOf:
номер экспоненты	indexOfExponent
номер элементарного метода:	primitiveIndexOf:

О

обменять указатель: и:	swapPointersOf: and:
обновить:	update:
обратить биты	bitInvert
обратное	reciprocal
общее поступление из:	totalReceivedFrom:
общие траты на:	totalSpentFor:
объединить:	merge:
объект от:	objectAt:
объект от: пом:	objectAt: put:
объект целое для:	integerObjectOf:

округлить	rounded
округлить до:	roundTo:
окружение	environment
операнд:	operand:
определение экземпляра:	instanceSpecificationOf:
определяемый термин	definitionTerm
определяемый термин:	definitionTerm:
оригинал для:	originalFor:
оригинал для временной переменной:	originalForTempVariable:
оригинал для временной переменной: в методе:	originalForTempVariable: inMethod:
оригинал для глобальной переменной:	originalForGlobalVariable:
оригинал для переменной экземпляра:	originalForInstanceVariable:
оригинал или пусто для:	originalOrNilFor:
оригинальное имя для глобальной переменной:	originalNameForGlobalVariable:
оригинальный символ	originalSymbol
оригинальный язык для:	originalLanguageFor:
освободить	release
основание	radix
основание:	radix:
основной новый	basicNew
основной новый:	basicNew:
основной от:	basicAt:
основной от: пом:	basicAt: put:
основной размер	basicSize
останов	halt
остаток	rest
от:	at:
от: до:	from: to:
от: до: через:	from: to: by:
от: если есть:	at: ifPresent:
от: если нету:	at: ifAbsent:
от: пом:	at: put:

от всех: пом:	atAll: put:
от всех: пом все:	atAll: putAll:
от всех пом:	atAllPut:
от нового номера: пом:	atNewIndex: put:
отбросить:	reject:
ответ элементарного метода	primitiveResponse
ответ элементарного метода специального селектора	specialSelectorPrimitiveResponse
ответственность подкласса	subclassResponsibility
отвечает на:	respondsTo:
откомпилированный метод от:	compiledMethodAt:
открыть	open
отменить выталкивание:	unPop:
отправитель	sender
отрицательное	negative
очистить	clear
очистить поля контекста	nilContextFields
ошибка	error
ошибка:	error:
ошибка границы номера:	errorSubscriptBounds:
ошибка значение не найдено	errorValueNotFound
ошибка ключ не найден	errorKeyNotFound
ошибка не ключевой	errorNotKeyed
ошибка не найден:	errorNotFound:
ошибка нету такого элемента	errorNoSuchElement

П

параграф	paragraph
параграф:	paragraph:
пер экз от:	instVarAt:
пер экз от: пом:	instVarAt: put:
первый	first
первый контекст	firstContext
переведённый	translated
перевести: с: на: как:	translate: from: to: as:

перевести имена переменных экземпляра на:	translateInstanceVariableNamesIn:
перевести на: с:	translateIn: with:
перевести на: с: класс:	translateIn: with: class:
перевести на: часть селектора:	translateIn: selectorPart:
перевести переменные класса как:	translateClassVariablesAs:
перевести переменные метода	translateMethodVariables
перевести переменные метода: как:	translateVariablesForMethod: as:
перевести переменные на: с: класс:	translateVariablesIn: with: class:
перевести переменные экземпляра как:	translateInstanceVariablesAs:
перевести селектор	translateSelector
перевести селектор на:	translateSelectorIn:
перевести селекторы на:	translateSelectorsIn:
перевести символы	translateSymbols
перевести символы как:	translateSymbolsAs:
перевести символы на:	translateSymbolsIn:
перевести часть используются на:	translatePartUsesIn:
перевести числа на:	translateNumbersIn:
перевод для: на:	translationFor: into:
перевод для селектора: на:	translationForSelector: into:
перевод специальной переменной: на:	translationForSpecialVariable: into:
перед:	before:
переименовать:	rename:
перейти к:	transferTo:
переключиться и установить шрифт для ИД:	switchAndInstallFontToID:
перекомпилировать:	recompile:
переменная	variable
переменная:	variable:
переменная: значение:	variable: value:

переменный подкласс: имена переменных экземпляра: имена переменных класса: словари пула: категория:	variableSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:
переменный подкласс байтов: имена переменных экземпляра: имена переменных класса: словари пула: категория:	variableByteSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:
переменный подкласс слов: имена переменных экземпляра: имена переменных класса: словари пула: категория:	variableWordSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:
перенести: от номера: из объекта: в номер: в объект:	transfer: fromField: ofObject: toField: ofObject:
пересекает:	intersects:
пересечь:	intersect:
печатать в:	printOn:
печатать в: заголовок с аргументами:	printOn: headerWithArguments:
печатать в: новую строку с отступом:	printOn: newLineWithIdent:
печатать в: предложения:	printOn: statements:
печатать временные в:	printTemporariesOn:
печатать имена переменных класса в: на языке:	printClassVariableNamesOn: language:
печатать имена переменных класса в: селектор: язык:	printClassVariableNamesOn: selector: language:
печатать имена переменных экземпляра в: на языке:	printInstanceVariableNamesOn: language:
печатать имена переменных экземпляра в: селектор: язык:	printInstanceVariableNamesOn: selector: language:
печатать имя в:	printNameOn:
печатать категорию в: на языке:	printCategoryOn: inLanguage:
печатать комментарий в:	printCommentOn:

печатать получателя в: отступ: в одну строку:	printReceiverOn: ident: inOneLine:
печатать разделяемые пулы в: на языке:	printSharedPoolsOn: inLanguage:
печатать ряд в:	printArrayOn:
печатать селектор и аргументы в: отступ: в одну строку:	printSelectorAndArgumentsOn: ident: inOneLine:
печатать число в:	printNumberOn:
печатать элементы в:	printElementsOn:
пи	pi
побитовое и:	bitAnd:
побитовое или:	bitOr:
побитовое искл или:	bitXor:
поверхностная копия	shallowCopy
подкласс:	subclass:
подкласс: имена переменных экземпляра: имена переменных класса: словари пула: категория:	subclass: instanceVariableNames: classVariableNames: poolDictionaries: category:
подкласс: используются: имена переменных экземпляра: имена переменных класса: словари пула: категория:	subclass: uses: instanceVariableNames: classVariableNames: poolDictionaries: category:
подкласс доступа: имена переменных экземпляра: имена переменных класса: словари пула: категория:	accessingSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:
подклассы	subclasses
подномер: с:	subscript: with:
подномер: с: сохранить:	subscript: with: storing:
позиция	position
пока истина	whileTrue
пока истина:	whileTrue:
пока ложь	whileFalse
пока ложь:	whileFalse:
показать	display
показать:	show:

показать в:	displayAt:
пол	floor
положение	location
положительное	positive
положительное 16 битное значение для:	positive16BitValueOf:
положительное 16 битное целое для:	positive16BitIntegerFor:
получатель	receiver
получатель:	receiver:
получатель: селектор: аргументы:	receiver: selector: arguments:
получатель: сообщения:	receiver: messages:
получить: из:	receive: from:
пом следующим:	nextPut:
пом следующим словом:	nextWordPut:
пом следующими все:	nextPutAll:
пом следующими цепь:	nextStringPut:
поместить:	store:
поместить в:	storeOn:
поместить временную переменную:	pushTemporaryVariable:
поместить литерал константу:	pushLiteralConstant:
поместить литерал переменную:	pushLiteralVariable:
поместить переменную получателя:	pushReceiverVariable:
послать должен быть логическим	sendMustBeBoolean
послать наду	superSend
послать наду:	superSend:
послать селектор: количество аргументов:	sendSelector: argumentCount:
послать селектор классу:	sendSelectorToClass:
после:	after:
последний	last
потолок	ceiling

потратить: на:	spend: for:
потратить: на: вычесть:	spend: for: deducting:
потратить на налоги: на:	spendDeductible: for:
правый	right
правый:	right:
правый центр	rightCenter
прагмы:	pragmas:
предложения	statements
предложения:	statements:
преобразовать для выполнения	convertToDoIt
прим сигнал: при миллисекундах:	primSignal: atMilliseconds:
применить:	apply:
приоритет	priority
приоритет:	priority:
приоритет аргумента:	argumentPrecedence:
приоритет интерфейса с пользователем	userSchedulingPriority
приоритет получателя	receiverPrecedence
приоритет прерываний пользователя	userInterruptPriority
приоритет синхронных процессов	timingPriority
приоритет узла:	nodePriority:
приоритет фона пользователя	userBackgroundPriority
приоритет фона системы	systemBackgroundPriority
приостановить	suspend
приостановить активный	suspendActive
присвоить данные:	setData:
присвоить карты	setCards
присвоить набор:	setCollection:
присвоить набор: карту:	setCollection: map:
присвоить от: до: через:	setFrom: to: by:
присвоить переменные	setVariables
присвоить содержимое:	setContents:
присвоить элементу:	setElement:

пробел	space
проверить:	verify:
проверить в:	verifyIn:
проверить границы нумерования для: в:	checkIndexableBoundsOf: in:
проверить границы переменных экземпляра для: в:	checkInstanceVariableBoundsOf: in:
проверить на пустость	emptyCheck
проверить переключение процессов	checkProcessSwitch
проверить число	verifyNumber
промежуток	span
пропустить:	skip:
пропустить до:	skipTo:
просмотреть для:	scanFor:
проснуться	wakeup
просто вернуть значение: к:	simpleReturnValue: to:
протолкнуть:	push:
протолкнуть целое:	pushInteger:
прыгнуть:	jump:
прыгнуть если: на:	jumplf: by:
пс	cr
пс таб	crtab
пс таб:	crtab:
пул класса	classPool
пустая копия	copyEmpty
пусто	nil
пусто:	ifNil:
пусто: не пусто:	ifNil: ifNotNil:
пустой	isEmpty
пятый	fifth

Р

Ряд констант	ArrayConstant
радианы в градусы	radiansToDegrees

раз повторить:	timesRepeat:
разбудить процесс с наивысшим приоритетом	wakeHighestPriority
разветвить	fork
разветвить от:	forkAt:
раздел	div
раздел:	div:
разделитель	separator
разделяемые пулы	sharedPools
размер	size
размер для:	sizeFor:
размер сообщения	messageSize
размеры	extent
размеры:	extent:
разновидность	species
разрыв	break
расти	grow
расширение заголовка:	headerExtensionOf:
расширенный байткод посылки	extendedSendBytecode
расширенный байткод сохранить	extendedStoreBytecode
расширенный байткод сохранить и вытолкнуть	extendedStoreAndPopBytecode
реверсивно делать:	reverseDo:
результат	result
результат 1	result1
результат 2	result2
результат 3	result3
родитель:	parent:
родительский словарь:	parentDictionary:
ряд переменных класса на:	classVariablesArrayIn:

С

Символ	Symbol

с:	with:
с: делать:	with: do:
с: за:	of: at:
с: от: до:	with: from: to:
с: с:	with: with:
с: с: с:	with: with: with:
с: с: с: с:	with: with: with: with:
с: с: с: с: с:	with: with: with: with: with:
с: с: с: с: с: с:	with: with: with: with: with: with:
с именем:	named:
с хвостом	withTail
сам	self
сбросить	reset
связать с:	bindWith:
сдвинуть биты:	bitShift:
себя	yourself
сегодня	today
седьмой	seventh
селектор	selector
селектор:	selector:
селектор: блок: прагмы: комментарий:	selector: block: pragmas: comment:
селектор: блок: прагмы: комментарий: хвост:	selector: block: pragmas: comment: tail:
селектор: язык:	selector: language:
селектор как символ	selectorAsSymbol
селектор метода	methodSelector
селекторы	selectors
сигнал	signal
символы	symbolsDictionary
син	sin
синхронный сигнал:	synchronousSignal:
ск	pc
ск:	pc:
следующая связь	nextLink
следующая связь:	nextLink:

следующая цепь	nextString
следующее пробуждение	nextWakeup
следующее слово	nextWord
следующее число:	nextNumber:
следующий	next
следующий:	next:
следующий за:	nextTo:
следующий совпадает с:	nextMatchFor:
следующий экземпляр	nextInstance
следующим числом: пом:	nextNumber: put:
следующими: пом:	next: put:
словари пула:	poolDictionaries:
словарь	dictionary
словарь методов	methodDictionary
словарь методов:	methodDictionary:
словарь селекторов	selectorsDictionary
слово	word
слово:	word:
сменить язык	changeLanguage
со всеми:	withAll:
со всеми подклассами	withAllSubclasses
со стилем для: делать:	withStyleFor: do:
собрать:	collect:
содержание потока:	streamContents:
содержащий:	intern:
содержащий знак:	internCharacter:
содержимое	contents
содержимое:	contents:
содержит:	includes:
содержит ассоциацию:	includesAssociation:
содержит ключ:	includesKey:
содержит о	includeR
содержит объект:	hasObject:
содержит селектор:	includesSelector:
содержит точку:	containsPoint:
содержит экземпляр:	containsInstanceOf:
создать пространство в конце	makeRoomAtLast

создать текущее сообщение	createActualMessage
сон:	sleep:
сообщения:	messages:
сопоставить с:	match:
сортированный	sort
сортирующий блок:	sortBlock:
сохранить байт: в объект: со значением:	storeByte: ofObject: withValue:
сохранить значение указателя инструкции: в контекст:	storeInstructionPointerValue: inContext:
сохранить значение указателя стэка: в контекст:	storeStackPointerValue: inContext:
сохранить регистры контекста	storeContextRegisters
сохранить слово: в объект: со значением:	storeWord: ofObject: withValue:
сохранить указатель: в объект: со значением:	storePointer: ofObject: withValue:
сохранить целое: в объект: со значением:	storeInteger: ofObject: withValue:
список определений	definitionList
список определений:	definitionList:
становится:	become:
старший байт:	highByteOf:
старший бит	highBit
старшинство	precedence
старшинство:	precedence:
стиль:	style:
строго положительное	strictlyPositive
строка таблицы	tableRow
строка таблицы:	tableRow:
существует перевод для:	thereIsTranslationFor:
существует перевод для: на:	thereIsTranslationFor: into:
счётчик команд цели	targetPC
считать	peek
считать для:	peekFor:

Т

таб	tab
таблица:	table:
такой же как:	sameAs:
тан	tan
тасовать	shuffle
текст:	text:
текущее	now
текущие дата и время	dateAndTimeNow
тень:	shade:
тест	test
тип:	type:
тип: номер:	type: index:
тип и номер переменной:	typeAndIndexOfVariable:
тип получателя	receiverType
третий	third
тэг:	tag:

У

увеличить колво врем пер	incAmountTempVars
увеличить ссылки на:	increaseReferencesTo:
угол	corner
угол:	corner:
удалить:	remove:
удалить: если нету:	remove: ifAbsent:
удалить все:	removeAll:
удалить зависимость:	removeDependent:
удалить из системы	removeFromSystem
удалить имя переменной класса:	removeClassVarName:
удалить имя переменной экземпляра:	removeInstVarName:
удалить категорию:	removeCategory:
удалить ключ:	removeKey:

удалить ключ: если нету:	removeKey: ifAbsent:
удалить номер:	removeIndex:
удалить первую связь списка:	removeFirstLinkOfList:
удалить первый	removeFirst
удалить подкласс:	removeSubclass:
удалить последний	removeLast
удалить разделяемый пул:	removeSharedPool:
удалить селектор:	removeSelector:
узкий пробел	thinSpace
указатель инструкции контекста:	instructionPointerOfContext:
указатель на Планировщика	schedulerPointer
указатель стэка контекста:	stackPointerOfContext:
уменьшить ссылки на:	decreaseReferencesTo:
умножить на два в степени:	timesTwoPower:
упорядоченный список	orderedList
упорядоченный список:	orderedList:
упростить	simplify
уровень:	level:
усечь	truncated
усечь до:	truncateTo:
успех	success
успех:	success:
установить в конец	setToEnd
установить на:	setOn:
устранить конфликт в:	fixCollisionsFrom:
уступить	yield
утверждение:	assert:

Ф

файл только для чтения с именем:	readOnlyFileNamed:
факториал	factorial
фигурный поток:	braceStream:
фигурный ряд	braceArray

фиксированных полей:	fixedFieldsOf:
флаг большого контекста:	largeContextFlagOf:
форма	form
форма:	form:
формат:	format:

Х

хвост:	tail:
хэш	hash
хэш:	hash:

Ц

Цепь	String
цветная печать аргументов в: отступ: в одну строку:	colorPrintArgumentsOn: ident: inOneLine:
цветная печать в:	colorPrintOn:
цветная печать в: отступ:	colorPrintOn: ident:
цветная печать в: отступ: в одну строку:	colorPrintOn: ident: inOneLine:
цветная печать в: шаблона для перевода переменных на: имя словарь:	colorPrintOn: templateForTranslateVariablesIn: dictionaryName:
цветная печать временных в: отступ: в одну строку:	colorPrintTemporariesOn: ident: inOneLine:
цветная печать предложений в: отступ: в одну строку:	colorPrintStatementsOn: ident: inOneLine:
целая часть	integerPart
цель:	target:
цель: тип:	target: type:
центр	center
цепь	string
цепь бирка	labelString
цепь для печати	printString

цепь для помещения	storeString
цепь переменных класса	classVariablesString
цепь переменных класса на:	classVariablesStringIn:
цепь переменных экземпляра	instanceVariablesString
цепь переменных экземпляра на:	instanceVariablesArrayIn:
цепь разделяемых пулов	sharedPoolsString
цикл	cycle
цифра от:	digitAt:
цифра от: пом:	digitAt: put:

Ч

Число	Number
чётное	even
частное:	quo:
часы	hours
четвёртый	fourth
числитель: знаменатель:	numerator: denominator:
читать из:	readFrom:

Ш

шаблон для перевода символов на: словарь:	templateForTranslateSymbolsIn: dictionary:
шаблон перевода символов для метода: в:	templateForTranslateSymbolsForMethod: in:
шестой	sixth
ширина	width

Э

экв:	eqv:
экземпляр	instance

экземпляр блока	blockCopy
экземпляр блока:	blockCopy:
экземпляр класса: с байтами:	instantiateClass: withBytes:
экземпляр класса: с указателями:	instantiateClass: withPointers:
экземпляр класса: со словами:	instantiateClass: withWords:
экземпляр после:	instanceAfter:
экземпляры для:	instancesOf:
эксп	exp
экспонента	exponent
элемент	element
элемент списка	listItem
элемент списка:	listItem:
элементарный больше или равно	primitiveGreaterOrEqual
элементарный больше чем	primitiveGreaterThan
элементарный в конце	primitiveAtEnd
элементарный ввести слово	primitiveInputWord
элементарный возобновить	primitiveResume
элементарный выйти	primitiveQuit
элементарный выйти в отладчик	primitiveExitToDebugger
элементарный выполнить	primitivePerform
элементарный выполнить с аргументами	primitivePerformWithArgs
элементарный вычесть	primitiveSubtract
элементарный вычесть плавающее	primitiveFloatSubtract
элементарный деление по модулю	primitiveMod
элементарный добавить	primitiveAdd
элементарный добавить плавающее	primitiveFloatAdd
элементарный дробная часть	primitiveFractionalPart
элементарный ждать	primitiveWait
элементарный заменить цепь	primitiveStringReplace
элементарный значение	primitiveValue

элементарный значение с аргументами	primitiveValueWithArgs
элементарный интервал семафора	primitiveSampleInterval
элементарный как объект	primitiveAsObject
элементарный как плавающее	primitiveAsFIoat
элементарный как УО	primitiveAsOop
элементарный копировать биты	primitiveCopyBits
элементарный меньше или равно	primitiveLessOrEqual
элементарный меньше чем	primitiveLessThan
элементарный метод арифметический селектор	arithmeticSelectorPrimitive
элементарный метод класс	primitiveClass
элементарный метод общий селектор	commonSelectorPrimitive
элементарный не равно	primitiveNotEqual
элементарный некоторый экземпляр	primitiveSomeInstance
элементарный новый	primitiveNew
элементарный новый метод	primitiveNewMethod
элементарный новый с аргументом	primitiveNewWithArg
элементарный Объект от	primitiveObjectAt
элементарный Объект от пом	primitiveObjectAtPut
элементарный оставшаяся память	primitiveCoreLeft
элементарный оставшиеся УО	primitiveOopsLeft
элементарный от	primitiveAt
элементарный от пом	primitiveAtPut
элементарный очистить кэш	primitiveFlushCache
элементарный пер экз от	primitiveInstVarAt
элементарный пер экз от пом	primitiveInstVarAtPut
элементарный плавающее больше или равно	primitiveFIoatGreaterOrEqual

элементарный плавающее больше чем	primitiveFloatGreaterThan
элементарный плавающее меньше или равно	primitiveFIoatLessOrEqual
элементарный плавающее меньше чем	primitiveFIoatLessThan
элементарный плавающее не равно	primitiveFIoatNotEqual
элементарный плавающее равно	primitiveFIoatEqual
элементарный плавающее разделить	primitiveFIoatDivide
элементарный плавающее умножить	primitiveFIoatMultiply
элементарный плавающее усечь	primitiveTruncated
элементарный побитовое и	primitiveBitAnd
элементарный побитовое или	primitiveBitOr
элементарный побитовое исключающее или	primitiveBitXor
элементарный пом следующим	primitiveNextPut
элементарный поместить курсор в положение	primitiveCursorLocPut
элементарный привязать курсор	primitiveCursorLink
элементарный приостановить	primitiveSuspend
элементарный просмотреть знаки	primitiveScanCharacters
элементарный равно	primitiveEqual
элементарный разд	primitiveDiv
элементарный разделить	primitiveDivide
элементарный размер	primitiveSize
элементарный сдвинуть биты	primitiveBitShift
элементарный сделать снимок	primitiveSnapshot
элементарный семафор ввода	primitiveInputSemaphore
элементарный сигнал	primitiveSignal

элементарный сигналить на тике	primitiveSignalAtTick
элементарный сигналить при оставшихся УО при оставшихся словах	primitiveSignalAtOopsLeftWordsLeft
элементарный следующий	primitiveNext
элементарный следующий экземпляр	primitiveNextInstance
элементарный слова времени в	primitiveTimeWordsInto
элементарный слова тиков в	primitiveTickWordsInto
элементарный создать точку	primitiveMakePoint
элементарный становится	primitiveBecome
элементарный стать курсором	primitiveBeCursor
элементарный стать экраном	primitiveBeDisplay
элементарный точка мыши	primitiveMousePoint
элементарный умножить	primitiveMultiply
элементарный умножить на два в степени	primitiveTimesTwoPower
элементарный Цепь от	primitiveStringAt
элементарный Цепь от пом	primitiveStringAtPut
элементарный цикл отрисовки	primitiveDrawLoop
элементарный частное	primitiveQuo
элементарный эквивалентен	primitiveEquivalent
элементарный экземпляр блока	primitiveBlockCopy
элементарный экспонента	primitiveExponent
элементы:	elements:
это байты	isBytes
это биты	isBits
это буква	isLetter
это в верхнем регистре	isUppercase
это в нижнем регистре	isLowercase
это знак	isCharacter
это значение целого:	isIntegerValue:
это истина	isTrue
это контекст блока:	isBlockContext:
это лист	isLeaf

это ложь	isFalse
это над	isSuper
это нумерованный:	isIndexable:
это объект целое:	isIntegerObject:
это оригинал	isOriginal
это получатель	isReceiver
это последний экземпляр:	isLastInstance:
это пусто	isNil
это пустой список:	isEmptyList:
это разновидность:	isKindOf:
это сам	isSelf
это селектор из букв:	isLetterSelector:
это слова	isWords
это слова:	isWords:
это специальная переменная	isSpecialVariable
это указатели	isPointers
это указатели:	isPointers:
это цифра	isDigit
это цифра: основание:	isDigit: radix:
это член:	isMemberOf:
это этот контекст	isThisContext

Я

язык	language
язык:	language:
язык ИСО	isoLanguage:
язык компилятора по умолчанию	defaultCompilerLanguage
якорь	anchor
яргт:	html:

Глава 33

Словари имён классов

Оглавление

33.1 Английско-русский словарь имён классов

A

ArrayedCollection	Набор ряд
Array	Ряд
Association	Ассоциация

B

Bag	Мешок
Behavior	Поведение
BitBlt	ПерБлБит
BlockClosure	Блок замыкание

BlockContext	Контекст блока
BlockScope	Область видимости блока
Boolean	Логика
Browser	Браузер
ByteArray	Ряд байтов

C

Character	Знак
CharacterScanner	Сканер знаков
ClassDescription	Описание класса
Class	Класс
Collection	Набор
Color	Цвет
ColoredCodeStream	Поток цветного текста
CompiledMethod	Откомпилированный метод
ContextPart	Часть контекста
Cursor	Курсор

D

Date	Дата
DeductibleHistory	Налоговая история
Delay	Задержка
Dictionary	Словарь
DisplayScreen	Показываемый экран
DualListDictionary	Словарь двойной список
DuplicateStackTop	Удвоить вершину стека

E

Entry	Запись

F

False	Ложь
FastDictionary	Быстрый словарь
FileStream	Поток файла
Float	Плавающее
Four	Четыре
Fraction	Дробь

H

HF	Домашнее хозяйство

I

IdentityDictionary	Тождественный словарь
InputSensor	Датчик ввода
Integer	Целое
Interval	Интервал

J

JumpCode	Код перехода

L

LargeNegativeInteger	Большое отрицательное целое
LargePositiveInteger	Большое положительное целое
LexicalScope	Область видимости
LinkedListStream	Поток связанного списка
Link	Связь
LinkedList	Связанный список
LookupKey	Ключ поиска

M

M17nBrowser	Многоязычный браузер
Magnitude	Величина
MappedCollection	Набор отображение
Message	Сообщение
Metaclass	Метакласс
MethodContext	Контекст метода
MethodDictionary	Словарь методов
MethodProperties	Свойства метода

N

NameOfSubclassTest2	ИмяПодклассаТест2
NameOfSubclassTest3	Имя класса тест 3
Node	Узел
Number	Число

O

Object	Объект
One	Один
OrderedCollection	Упорядоченный набор
OriginalSelectorTranslation	Перевод селектора оригинала

P

PersonnelRecord	Запись о человеке
Point	Точка
PopAndStore	Извлечь и запомнить
PopStackTop	Извлечь вершину стека
PositionableStream	Позиционируемый поток
Preference	Предпочтение
Preferences	Предпочтения

Process	Процесс
Processor	Исполнитель
ProcessorScheduler	Планировщик исполнителя
Product	Продукт
ProtoObject	Прото объект
PushActiveContext	Поместить текущий контекст
PushLiteralConstant	Поместить константу литерал
PushSpecialValue	Поместить специальное значение
PushVariable	Поместить переменную

R

Random	Случайное число
ReadStream	Поток чтения
ReadWriteStream	Поток чтения записи
Rectangle	Прямоугольник
ReturnCode	Код возврата
RunArray	Ряд серий

S

SampleSpaceWithout-Replacement	Пространство выбора без замены
SampleSpaceWithReplacement	Пространство выбора с заменой
SelectorTranslation	Перевод селектора
Semaphore	Семафор
Send	Посылка сообщения
SendSpecialMessage	Послать специальное сообщение
SequenceableCollection	Набор последовательность
Set	Множество
SharedQueue	Разделяемая очередь
SmallDictionary	Малый словарь

SmallInteger	Малое целое
Smalltalk	Смолток
SortedCollection	Сортированный набор
SourceDictionary	Словарь исходников
SourceMethod	Исходный метод
SqNumberParser	Анализатор чисел
SqueakM17nDictionary	Многоязычный словарь
STAssignmentNode	ИД узел присваивание
STBlockNode	ИД узел блок
STBlockVariableNode	ИД узел переменная блока
STBraceNode	ИД узел фигурные скобки
STCascadeNode	ИД узел каскад
STLiteralNode	ИД узел литерал
STMessageNode	ИД узел сообщение
STMethodNode	ИД узел метод
STNode	ИД узел
STPipeNode	ИД узел труба
STReturnNode	ИД узел возврат
STSpecialVariableNode	ИД узел специальная переменная
STTailNode	ИД узел хвост
STTemporaryVariableNode	ИД узел временная переменная
STVariableNode	ИД узел переменная
Store	Запомнить
Stream	Поток
String	Цепь
Symbol	Символ
SystemDictionary	Словарь системы

T

Text	Текст
Three	Три
Tile	Плитка
Time	Время

Transcript	Транскрипт
True	Истина
Two	Два

U

UndefinedObject	Неопрделённый объект

V

VMInstruction	Инструкция ВМ

W

WordLink	Связь слово
WriteStream	Поток записи

33.2 Русско-английский словарь имён классов

А

Анализатор чисел	SqNumberParser
Ассоциация	Association

Б

Блок замыкание	BlockClosure
Большое отрицательное целое	LargeNegativeInteger
Большое положительное целое	LargePositiveInteger
Браузер	Browser
Быстрый словарь	FastDictionary

В

Величина	Magnitude
Время	Time

Д

Дата	Date
Датчик ввода	InputSensor
Два	Two
Домашнее хозяйство	HF
Дробь	Fraction

З

Задержка	Delay
Запись о человеке	PersonnelRecord
Запись	Entry
Запомнить	Store
Знак	Character

И

ИД узел блок	STBlockNode
ИД узел возврат	STReturnNode
ИД узел временная переменная	STTemporaryVariableNode
ИД узел каскад	STCascadeNode
ИД узел литерал	STLiteralNode
ИД узел метод	STMethodNode
ИД узел переменная блока	STBlockVariableNode
ИД узел переменная	STVariableNode
ИД узел присваивание	STAssignmentNode
ИД узел сообщение	STMessageNode

ИД узел специальная переменная	STSpecialVariableNode
ИД узел труба	STPipeNode
ИД узел фигурные скобки	STBraceNode
ИД узел	STNode
ИД узел хвост	STTailNode
Извлечь вершину стека	PopStackTop
Извлечь и запомнить	PopAndStore
Имя класса тест 3	NameOfSubclassTest3
ИмяПодклассаТест2	NameOfSubclassTest2
Инструкция ВМ	VMInstruction
Интервал	Interval
Исполнитель	Processor
Истина	True
Исходный метод	SourceMethod

К

Класс	Class
Ключ поиска	LookupKey
Код возврата	ReturnCode
Код перехода	JumpCode
Контекст блока	BlockContext
Контекст метода	MethodContext
Курсор	Cursor

Л

Логика	Boolean
Ложь	False

М

Малое целое	SmallInteger

Малый словарь	SmallDictionary
Метакласс	Metaclass
Мешок	Bag
Многоязычный браузер	M17nBrowser
Многоязычный словарь	SqueakM17nDictionary
Множество	Set

Н

Набор отображение	MappedCollection
Набор последовательность	SequenceableCollection
Набор	Collection
Набор ряд	ArrayedCollection
Налоговая история	DeductibleHistory
Неопрделённый объект	UndefinedObject

О

Область видимости блока	BlockScope
Область видимости	LexicalScope
Объект	Object
Один	One
Описание класса	ClassDescription
Откомпилированный метод	CompiledMethod

П

ПерБлБит	BitBlt
Перевод селектора	SelectorTranslation
Перевод селектора оригинала	OriginalSelectorTranslation
Плавающее	Float
Планировщик исполнителя	ProcessorScheduler
Плитка	Tile
Поведение	Behavior

Позиционируемый поток	PositionableStream
Показываемый экран	DisplayScreen
Поместить константу литерал	PushLiteralConstant
Поместить переменную	PushVariable
Поместить специальное значение	PushSpecialValue
Поместить текущий контекст	PushActiveContext
Послать специальное сообщение	SendSpecialMessage
Посылка сообщения	Send
Поток записи	WriteStream
Поток	Stream
Поток связанного списка	LinkedListStream
Поток файла	FileStream
Поток цветного текста	ColoredCodeStream
Поток чтения записи	ReadWriteStream
Поток чтения	ReadStream
Предпочтение	Preference
Предпочтения	Preferences
Продукт	Product
Пространство выбора без замены	SampleSpaceWithout-Replacement
Пространство выбора с заменой	SampleSpaceWithReplacement
Прото объект	ProtoObject
Процесс	Process
Прямоугольник	Rectangle

Р

Разделяемая очередь	SharedQueue
Ряд	Array
Ряд байтов	ByteArray
Ряд серий	RunArray

С

Свойства метода	MethodProperties
Связанный список	LinkedList
Связь слово	WordLink
Связь	Link
Семафор	Semaphore
Символ	Symbol
Сканер знаков	CharacterScanner
Словарь двойной список	DualListDictionary
Словарь исходников	SourceDictionary
Словарь методов	MethodDictionary
Словарь	Dictionary
Словарь системы	SystemDictionary
Случайное число	Random
Смолток	Smalltalk
Сообщение	Message
Сортированный набор	SortedCollection

Т

Текст	Text
Тождественный словарь	IdentityDictionary
Точка	Point
Транскрипт	Transcript
Три	Three

У

Удвоить вершину стека	DuplicateStackTop
Узел	Node
Упорядоченный набор	OrderedCollection

Ц

Цвет	Color
Целое	Integer
Цепь	String

Ч

Часть контекста	ContextPart
Четыре	Four
Число	Number