Программирование на языке RUBY

Идеология языка, теория и практика применения

Хэл Фултон

ДМК Пресс издательство

Профессиональная серия от Addison-Wesley

УДК 004.438
ББК 32.973.26-018.2

Фултон Х.
Ф94 Программирование на языке Ruby. – М.: ДМК Пресс, 2007. – 688 с.: ил.

ISBN 5-94074-357-9

Ruby – относительно новый объектно-ориентированный язык, разработанный Юкихиро Мацумото в 1995 году и позаимствовавший некоторые особенности у языков LISP, Smalltalk, Perl, CLU и других. Язык активно развивается и применяется в самых разных областях: от системного администрирования до разработки сложных динамических сайтов.

Книга является полноценным руководством по Ruby – ее можно использовать и как учебник, и как справочник, и как сборник ответов на вопросы типа «как сделать то или иное в Ruby». В ней приведено свыше 400 примеров, разбитых по различным аспектам программирования, и к которым автор дает обстоятельные комментарии.

Издание предназначено для программистов самого широкого круга и самой разной квалификации, желающих научиться качественно и профессионально работать на Ruby.

УДК 004.438
ББК 32.973.26-018.2

ISBN 0672328844
ISBN 5-94074-357-9

Содержание

Глава 1. Обзор Ruby

Язык формирует способ нашего мышления
и определяет то, о чем мы можем размышлять.
Бенджамин Ди Уорф

Стоит напомнить, что в новом языке программирования иногда видят панацею, особенно его адепты. Но ни один язык не сможет заменить все остальные. Не существует инструмента, безусловно пригодного для решения любой мыслимой задачи. Есть много предметных областей и много ограничений, налагаемых решаемыми в них задачами.

А самое главное – есть разные пути обдумывания задач, и это следствие разного опыта и личных качеств самих программистов. Поэтому в обозримой перспективе будут появляться все новые и новые языки. А пока есть много языков, будет много людей, которые их критикуют и защищают. Короче говоря, «языковым войнам» конца не предвидится, но мы в этой книге не станем принимать в них участия.

И тем не менее в постоянном поиске новой, более удачной системы записи программ нас иногда озаряют идеи, переживающие контекст, в котором зародились. Как Pascal многое позаимствовал у Algol, как Java выросла из C, так и каждый язык что-то берет у своих предшественников.

Язык – это одновременно набор инструментов и площадка для игр. У него есть практическая сторона, но он же служит и полигоном для испытания новых идей, которые могут быть приняты или отвергнуты сообществом программистов.

Одна из наиболее далеко идущих идей – концепция объектно-ориентированного программирования (ООП). Многие скажут, что значимость ООП имеет скорее эволюционный, нежели революционный характер, но никто не возразит против того, что оно оказало огромное влияние на индустрию. Двадцать пять лет назад объектная ориентированность представляла в основном академический интерес; сегодня это универсально принятая парадигма.

Вездесущность ООП породила много «рекламной чепухи» в индустрии. В классической работе, написанной в конце 1980-х годов, Роджер Кинг отметил: «Если вы хотите продать кошку специалисту по компьютерам, скажите, что она объектно-ориентированная». Мнения по поводу того, что на самом деле представляет собой ООП, весьма неоднородны, и даже среди тех, кто разделяет общую точку зрения, имеются разногласия относительно терминологии.

Мы не ставим себе целью поучаствовать в спорах. Мы согласны, что ООП – полезный инструмент и ценная методология решения задач; мы не утверждаем, что она способна излечить рак.

Что касается истинной природы ООП, то у нас есть любимые определения и термины, но мы пользуемся ими лишь для эффективного общения, так что пререкаться по поводу смысла слов не станем.

Обо всем этом пришлось сказать лишь потому, что знакомство с основами ООП необходимо для чтения этой книги и понимания примеров и подходов. Что бы ни говорили о Ruby, он безусловно является объектно-ориентированным языком.

1.1. Введение в объектно-ориентированное программирование

Прежде чем начать разговор о самом языке Ruby, неплохо было бы потолковать об объектно-ориентированном программировании вообще. Поэтому сейчас мы вкратце рассмотрим общие идеи, лишь слегка касаясь Ruby.

1.1.1. Что такое объект

В объектно-ориентированном программировании объект – фундаментальное понятие. *Объект* – это сущность, служащая контейнером для данных и управляющая доступом к этим данным. С объектом связан набор *атрибутов*, которые в сущности представляют собой просто переменные, принадлежащие объекту. (В этой книге мы будем без стеснения употреблять привычный термин «переменная» в применении к атрибутам.) Кроме того, с объектом ассоциирован набор функций, предоставляющих интерфейс к функциональным возможностям объекта. Эти функции называются *методами*.

Важно отметить, что любой объектно-ориентированный язык предоставляет механизм инкапсуляции. В общепринятом смысле это означает, во-первых, что атрибуты и методы объекта ассоциированы именно с этим объектом, а во-вторых, что область видимости атрибутов и методов по умолчанию ограничена самим объектом (применение принципа сокрытия информации).

Объект считается экземпляром класса объекта (обычно он называется просто *классом*). Класс можно представлять себе как чертеж или образец, а объект – как вещь, изготовленную по этому чертежу. Также класс часто называют абстрактным типом, то есть типом более сложным, нежели целое или строка символов.

Создание объекта (экземпляра класса) называют *инстанцированием*. В некоторых языках имеются явные конструкторы и деструкторы – функции, выполняющие действия, необходимые соответственно для инициализации и уничтожения объекта. Отметим попутно, что в Ruby есть нечто, что можно назвать конструктором, но никакого аналога деструктора не существует (благодаря наличию механизма сборки мусора).

Иногда возникает ситуация, когда некоторые данные имеют широкую область видимости, не ограниченную одним объектом, и помещать копию такого атрибута в каждый экземпляр класса неправильно. Рассмотрим, к примеру, класс `MyDogs` и

три объекта этого класса: `fido`, `rover` и `spot`. У каждой собаки могут быть такие атрибуты, как возраст и дата вакцинации. Предположим, однако, что нужно сохранить еще и имя владельца всех собак. Можно, конечно, поместить его в каждый объект, но это пустая трата памяти, к тому же искажающая смысл дизайна. Ясно, что атрибут «имя владельца» принадлежит не отдельному объекту, а классу в целом. Такие атрибуты (синтаксис их определения в разных языках различен) называются *атрибутами класса* (или *переменными класса*).

Есть немало ситуаций, в которых может понадобиться переменная класса. Допустим, например, что нужно знать, сколько всего было создано объектов некоторого класса. Можно было бы завести переменную класса, инициализировать ее нулем и увеличивать на единицу при создании каждого объекта. Эта переменная ассоциирована именно с классом, а не с каким-то конкретным объектом. С точки зрения области видимости она не отличается от любого другого атрибута, но существует лишь одна ее копия для всего множества объектов данного класса.

Чтобы отличить атрибуты класса от обыкновенных атрибутов, последние часто называют *атрибутами объекта* (или *атрибутами экземпляра*). Условимся, что в этой книге под словом «атрибут» понимается атрибут экземпляра, если явно не оговорено, что это атрибут класса.

Точно так же методы объекта служат для управления доступом к его атрибутам и предоставляют четко определенный интерфейс для этой цели. Но иногда желательно или даже необходимо определить метод, ассоциированный с самим классом. Неудивительно, что метод класса управляет доступом к переменным класса, кроме того, выполняя действия, распространяющиеся на весь класс, а не на какой-то конкретный объект. Как и в случае с атрибутами, мы будем считать, что метод принадлежит объекту, если явно не оговорено противное.

Стоит отметить, что в некотором смысле все методы являются методами класса. Не нужно думать, что, создав сто объектов, мы породили сотню копий кода методов! Однако правила ограничения области видимости гласят, что метод каждого объекта оперирует данными только того объекта, от имени которого вызван. Тем самым у нас создается иллюзия, будто методы объекта ассоциированы с самими объектами.

1.1.2. Наследование

Мы подходим к одной из самых сильных сторон ООП – наследованию. *Наследование* – это механизм, позволяющий расширять ранее определенную сущность путем добавления новых возможностей. Короче говоря, наследование – это способ повторного использования кода. (Простой и эффективный механизм повторного использования долго был Святым Граалем в информатике. Много десятилетий назад его поиски привели к изобретению параметризованных процедур и библиотек. ООП – лишь одна из последних попыток реализации искомого.)

Обычно наследование рассматривается на уровне класса. Если нам необходим какой-то класс, а в наличии имеется более общий, то можно определить свой класс так, чтобы он наследовал поведение уже существующего. Предположим, например, что есть класс `Polygon`, описывающий выпуклые многоугольники. Тогда класс

прямоугольника `Rectangle` можно унаследовать от `Polygon`. При этом `Rectangle` будет иметь все атрибуты и методы класса `Polygon`. Так, может уже быть написан метод, вычисляющий периметр путем суммирования длин всех сторон. Если все было реализовано правильно, этот метод автоматически будет работать и для нового класса; переписывать код не придется.

Если класс `B` наследует классу `A`, мы говорим, что `B` является подклассом `A`, а `A` – суперкласс `B`. По-другому говорят, что `A` – *базовый* или *родительский класс*, а `B` – *производный* или *дочерний класс*.

Как мы видели, производный класс может трактовать методы своего базового класса как свои собственные. С другой стороны, он может переопределить метод базового класса, предоставив иную его реализацию. Кроме того, в большинстве языков есть возможность вызвать из переопределенного метода метод базового класса с тем же именем. Иными словами, метод `foo` класса `B` знает, как вызвать метод `foo` класса `A`. (Любой язык, не предоставляющий такого механизма, можно заподозрить в отсутствии истинной объектной ориентированности.) То же верно и в отношении атрибутов.

Отношение между классом и его суперклассом интересно и важно, обычно его называют отношением «является». Действительно, квадрат `Square` «является» прямоугольником `Rectangle`, а прямоугольник `Rectangle` «является» многоугольником `Polygon` и т. д. Поэтому, рассматривая иерархию наследования (а такие иерархии в том или ином виде присутствуют в любом объектно-ориентированном языке), мы видим, что в любой ее точке специализированные сущности «являются» подклассами более общих. Отметим, что это отношение транзитивно, – если обратиться к предыдущему примеру, то квадрат «является» многоугольником. Однако отношение «является» не коммутативно – каждый прямоугольник есть многоугольник, но не каждый многоугольник – прямоугольник.

Это подводит нас к теме множественного наследования. Можно представить себе класс, который наследует нескольким классам. Например, классы `Dog` (Собака) и `Cat` (Кошка) могут наследовать классу `Mammal` (Млекопитающее), а `Sparrow` (Воробей) и `Raven` (Ворон) – классу `WingedCreature` (Крылатое). Но как быть с классом `Bat` (ЛетучаяМышь)? Он с равным успехом может наследовать и `Mammal`, и `WingedCreature`! Это хорошо согласуется с нашим жизненным опытом, ведь многие вещи можно отнести не к одной категории, а сразу к нескольким, не вложенным друг в друга.

Множественное наследование, вероятно, наиболее противоречивая часть ООП. Некоторые указывают на потенциальные неоднозначности, требующие разрешения. Например, если в обоих классах `Mammal` и `WingedCreature` имеется атрибут `size` (размер) или метод `eat` (есть), то какой из них имеется в виду, когда мы обращаемся к нему из объекта класса `Bat`? С этой трудностью тесно связана проблема ромбовидного наследования; она называется так из-за формы диаграммы наследования, возникающей, когда оба суперкласса наследуют одному классу. Представьте себе, что классы `Mammal` и `WingedCreature` наследуют общему предку `Organism` (Организм); тогда иерархия наследования от `Organism` к `Bat` будет иметь форму ромба. Но как быть с атрибутами, которые оба промежуточных класса наследуют от

своего родителя? Получает ли `Bat` две копии? Или они должны быть объединены в один атрибут, поскольку все равно заимствованы у общего предка?

Это скорее проблемы проектировщика языка, а не программиста. В разных объектно-ориентированных языках они решаются по-разному. Иногда вводятся правила, согласно которым какое-то одно определение атрибута «выигрывает». Либо же предоставляется возможность различать одноименные атрибуты. Иногда даже язык позволяет вводить псевдонимы или переименовывать идентификаторы. Многими это рассматривается как аргумент против множественного наследования – о механизмах разрешения подобных конфликтов имен нет единого мнения, поэтому все они «языкозависимы». В языке C++ предлагается минимальный набор средств для разрешения неоднозначностей; механизмы языка Eiffel, наверное, получше, а в Perl проблема решается совсем по-другому.

Есть и альтернатива – полностью запретить множественное наследование. Такой подход принят в языках Java и Ruby. На первый взгляд, это даже не назовешь компромиссным решением, но, вскоре мы убедимся, что все не так плохо, как кажется. Мы познакомимся с приемлемой альтернативой традиционному множественному наследованию, но сначала обсудим полиморфизм – еще одно понятие из арсенала ООП.

1.1.3. Полиморфизм

Термин «полиморфизм», наверное, вызывает самые жаркие семантические споры. Каждый знает, что это такое, но все понимают его по-разному. (Не так давно вопрос «Что такое полиморфизм?» стал популярным во время собеседования при поступлении на работу. Если его зададут вам, рекомендую процитировать какого-нибудь эксперта, например Бертрана Мейера или Бьерна Страуструпа; если собеседник не согласится, то пусть он спорит с классиком, а не с вами.)

Буквально слово «полиморфизм» означает «способность принимать разные формы или обличья». В самом широком смысле так называют ситуацию, когда различные объекты по-разному отвечают на одно и то же сообщение или вызов метода.

Дамиан Конвей (Damian Conway) в книге «Object-Oriented Perl» проводит смысловое различие между двумя видами полиморфизма. Первый, *наследственный полиморфизм*, – то, что имеет в виду большинство программистов, говорящих о полиморфизме.

Если некоторый класс наследует своему суперклассу, то по определению все методы суперкласса присутствуют также и в подклассе. Таким образом, цепочка наследования представляет собой линейную иерархию классов, отвечающих на одни и те же методы. Нужно, конечно, помнить, что в любом подклассе метод может быть переопределен; именно это и составляет сильную сторону наследования. При вызове метода объекта обычно отвечает либо метод, унаследованный от суперкласса, либо более специализированный вариант этого метода, созданный в интересах именно данного подкласса.

В языках со статической типизацией, например в C++, наследственный полиморфизм гарантирует совместимость типов вниз по цепочке наследования (но не

в обратном направлении). Скажем, если B наследует A, то указатель на объект класса A может указывать и на объект класса B; обратное же неверно. Совместимость типов – существенная черта ООП в подобных языках, можно даже сказать, что полиморфизм ей и исчерпывается. Но, конечно же, полиморфизм можно реализовать и в отсутствие статической типизации (как в Ruby).

Второй вид полиморфизма, упомянутый Конвеем, – это *интерфейсный полиморфизм*. Для него не требуется наличия отношения наследования между классами; нужно лишь, чтобы в интерфейсах объектов были методы с одним и тем же именем. Такие объекты можно трактовать как принадлежащие одному виду, и потому мы имеем некую разновидность полиморфизма (хотя в большинстве работ он так не называется).

Читатели, знакомые с языком Java, понимают, что в нем реализованы оба вида полиморфизма. Класс в Java может расширять другой класс, наследуя ему с помощью ключевого слова `extends`, а может с помощью ключевого слова `implements` реализовывать интерфейс, за счет чего приобретает заранее известный набор методов (которые необходимо переопределить). Такой синтаксис позволяет интерпретатору Java во время компиляции определить, можно ли вызывать данный метод для конкретного объекта.

Ruby поддерживает интерфейсный полиморфизм, но по-другому. Он позволяет определять модули, методы которых допускается «подмешивать» к существующим классам. Но обычно модули так не используются. Модуль состоит из методов и констант, которые можно использовать так, будто они являются частью класса или объекта. Когда модуль подмешивается с помощью предложения `include`, мы получаем ограниченную форму множественного наследования. (По словам проектировщика языка Юкихиро Мацумото, это можно рассматривать как одиночное наследование с разделением реализации.) Таким образом удается сохранить преимущества множественного наследования, не страдая от его недостатков.

1.1.4. Еще немного терминов

В языках, подобных C++, существует понятие *абстрактного класса*. Такому классу разрешается наследовать, но создать его экземпляр невозможно. В более динамичном языке Ruby такого понятия нет, но если программист пожелает, то может смоделировать его, потребовав, чтобы все методы были переопределены в производных классах. Полезно это или нет, оставляем на усмотрение читателя.

Создатель языка C++ Бьерн Страуструп определяет также понятие *конкретного типа*. Это класс, существующий только для удобства. Он спроектирован не для наследования; более того, ожидается, что ему никто никогда наследовать не будет. Другими словами, преимущества ООП в этом случае сводятся только к инкапсуляции. Ruby не поддерживает такой конструкции синтаксически (как и C++), но по природе своей прекрасно приспособлен для создания подобных классов.

Считается, что некоторые языки поддерживают более «чистую» модель ООП, чем другие. (К ним мы применяем термин «радикально объектно-ориентированный».) Это означает, что любая сущность в языке является объектом, даже примитивные типы представлены полноценными классами, а переменные и константы

рассматриваются как экземпляры. В таких языках, как Java, C++ и Eiffel, дело обстоит иначе. В них примитивные типы (особенно константы) не являются настоящими объектами, хотя иногда могут рассматриваться как таковые с помощью «классов-оберток». Вероятно, есть языки, которые более радикально объектно ориентированы, чем Ruby, но их немного.

Большинство объектно-ориентированных языков статично; методы и атрибуты, принадлежащие классу, глобальные переменные и иерархия наследования определяются во время компиляции. Быть может, самый сложный концептуальный переход заключается в том, что в Ruby все это происходит динамически. И определения, и даже порядок наследования можно задавать во время исполнения. Честно говоря, каждое объявление или определение исполняется во время работы программы. Помимо прочих достоинств, это позволяет избавиться от условной компиляции, и во многих случаях получается более эффективный код.

На этом мы завершаем беглую экскурсию в мир ООП. Мы старались последовательно применять введенные здесь термины на протяжении всей книги. Перейдем теперь к краткому обзору самого языка Ruby.

1.2. Базовый синтаксис и семантика Ruby

Выше мы отметили, что Ruby – настоящий динамический объектно-ориентированный язык.

Прежде чем переходить к обзору синтаксиса и семантики, упомянем некоторые другие его особенности.

Ruby – прагматичный (agile) язык. Он пластичен и поощряет частую переработку (рефакторинг), которая выполняется без особого труда.

Ruby – интерпретируемый язык. Разумеется, в будущем ради повышения производительности могут появиться и компиляторы Ruby, но мы считаем, что у интерпретатора много достоинств. Он не только позволяет быстро создавать прототипы, но и сокращает весь цикл разработки.

Ruby ориентирован на выражения. Зачем писать предложение, когда выражения достаточно? Это означает, в частности, что программа становится более компактной, поскольку общие части выносятся в отдельное выражение и повторения удается избежать.

Ruby – язык сверхвысокого уровня (VHLL). Один из принципов, положенных в основу его проектирования, заключается в том, что компьютер должен работать для человека, а не наоборот. Под «плотностью» Ruby понимают тот факт, что сложные, запутанные операции можно записать гораздо проще, чем в языках более низкого уровня.

Начнем мы с рассмотрения общего духа языка и некоторых применяемых в нем терминов. Затем вкратце обсудим природу программ на Ruby, а потом уже перейдем к примерам.

Прежде всего отметим, что программа на Ruby состоит из отдельных строк, – как в C, но не как в «древних» языках наподобие Фортрана. В одной строке может быть сколько угодно лексем, лишь бы они правильно отделялись пропусками.

В одной строке может быть несколько предложений, разделенных точками с запятой; только в этом случае точка с запятой и необходима. Логическая строка может быть разбита на несколько физических при условии, что все, кроме последней, заканчиваются обратной косой чертой или лексическому анализатору дан знак, что предложение еще не закончено. Таким знаком может, например, быть запятая в конце строки.

Главной программы как таковой (функции `main`) не существует; исполнение происходит сверху вниз. В более сложных программах в начале текста могут располагаться многочисленные определения, за которыми следует (концептуально) главная программа. Но даже в этом случае программа исполняется сверху вниз, так как в Ruby все определения исполняются.

1.2.1. Ключевые слова и идентификаторы

Ключевые (или зарезервированные) слова в Ruby обычно не применяются ни для каких иных целей. Вот их полный перечень:

```
BEGIN     END     alias     and     begin
break     case    class     def     defined?
do        else    elsif     end     ensure
false     for     if        in      module
next      nil     not       or      redo
rescue    retry   return    self    super
then      true    undef     unless  until
when      while   yield
```

Имена переменных и других идентификаторов обычно начинаются с буквы или специального модификатора. Основные правила таковы:

- имена локальных переменных (и таких псевдопеременных, как `self` и `nil`) начинаются со строчной буквы или знака подчеркивания _;
- имена глобальных переменных начинаются со знака доллара `$`;
- имена переменных экземпляра (принадлежащих объекту) начинаются со знака «собачки» `@`;
- имена переменных класса (принадлежащих классу) предваряются двумя знаками `@` (`@@`);
- имена констант начинаются с прописной буквы;
- в именах идентификаторов знак подчеркивания _ можно использовать наравне со строчными буквами;
- имена специальных переменных, начинающиеся со знака доллара (например, `$1` и `$/`), здесь не рассматриваются.

Примеры:

- локальные переменные `alpha`, `_ident`, `some_var`;
- псевдопеременные `self`, `nil`, `__FILE__`;
- константы `K6chip`, `Length`, `LENGTH`;
- переменные экземпляра `@foobar`, `@thx1138`, `@NOT_CONST`;
- переменные класса `@@phydeaux`, `@@my_var`, `@@NOT_CONST`;
- глобальные переменные `$beta`, `$B12vitamin`, `$NOT_CONST`.

1.2.2. Комментарии и встроенная документация

Комментарии в Ruby начинаются со знака решетки (#), находящегося вне строки или символьной константы, и продолжаются до конца строки:

```
x = y + 5   # Это комментарий.
# Это тоже комментарий.
print "# А это не комментарий."
```

Предполагается, что встроенная документация будет извлечена из программы каким-нибудь внешним инструментом. С точки зрения интерпретатора это обычный комментарий. Весь текст, расположенный между строками, которые начинаются с лексем `=begin` и `=end` (включительно), игнорируется интерпретатором (этим лексемам не должны предшествовать пробелы).

```
=begin
Назначение этой программы -
излечить рак
и установить мир во всем мире.
=end
```

1.2.3. Константы, переменные и типы

В Ruby переменные не имеют типа, однако объекты, на которые переменные ссылаются, тип имеют. Простейшие типы – это *символ*, *число* и *строка*.

Числовые константы интуитивно наиболее понятны, равно как и строки. В общем случае строка, заключенная в двойные кавычки, допускает интерполяцию выражений, а заключенная в одиночные кавычки интерпретируется почти буквально – в ней распознается только экранированная обратная косая черта.

Ниже показана «интерполяция» переменных и выражений в строку, заключенную в двойные кавычки:

```
a = 3
b = 79
puts "#{a} умноженное на #{b} = #{a*b}"   #  3 умноженное на 79 = 237
```

Более подробная информация о литералах (числах, строках, регулярных выражениях и т. п.) приведена в следующих главах.

Стоит упомянуть особую разновидность строк, которая полезна прежде всего в небольших сценариях, применяемых для объединения более крупных программ. Строка, выводимая программой, посылается операционной системе в качестве подлежащей исполнению команды, а затем результат выполненной команды подставляется обратно в строку. В простейшей форме для этого применяются строки, заключенные в обратные кавычки. В более сложном варианте используется синтаксическая конструкция `%x`:

```
'whoami'
'ls -l'
%x[grep -i meta *.html | wc -l]
```

Регулярные выражения в Ruby похожи на символьные строки, но используются по-другому. Обычно в качестве ограничителя выступает символ косой черты.

Синтаксис регулярных выражений в Ruby и Perl имеет много общего. Подробнее о регулярных выражениях см. главу 3.

Массивы в Ruby – очень мощная конструкция; они могут содержать данные любого типа. Более того, в одном массиве можно хранить данные разных типов. В главе 8 мы увидим, что все массивы – это экземпляры класса `Array`, а потому к ним применимы разнообразные методы. Массив-константа заключается в квадратные скобки. Примеры:

```
[1, 2, 3]
[1, 2, "застегни мне молнию на сапоге"]
[1, 2, [3,4], 5]
["alpha", "beta", "gamma", "delta"]
```

Во втором примере показан массив, содержащий целые числа и строки. В третьем примере мы видим вложенный массив, а в четвертом – массив строк. Как и в большинстве других языков, нумерация элементов массива начинается с нуля. Так, в последнем из показанных выше примеров элемент `"gamma"` имеет индекс 2. Все массивы динамические, задавать размер при создании не нужно.

Поскольку массивы строк встречаются очень часто (а набирать их неудобно), для них предусмотрен специальный синтаксис:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

Здесь не нужны ни кавычки, ни запятые; элементы разделяются пробелами. Если встречаются элементы, содержащие внутренние пробелы, такой синтаксис, конечно, неприменим.

Для доступа к конкретному элементу массива по индексу применяются квадратные скобки. Результирующее выражение можно получить или выполнить для него присваивание:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Еще одна «могучая» конструкция в Ruby – это хэш. Его также называют ассоциативным массивом или словарем. Хэш – это множество пар данных; обыкновенно он применяется в качестве справочной таблицы или как обобщенный массив, в котором индекс не обязан быть целым числом. Все хэши являются экземплярами класса `Hash`.

Хэш-константа, как правило, заключается в фигурные скобки, а ключи отделяются от значений символом `=>`. Ключ можно считать индексом для доступа к ассоциированному с ним значению. На типы ключей и значений не налагается никаких ограничений. Примеры:

```
{1=>1, 2=>4, 3=>9, 4=>16, 5=>25, 6=>36}
{"cat"=>"cats", "ox"=>"oxen", "bacterium"=>"bacteria"}
{"водород"=>1, "гелий"=>2, "углерод"=>12}
{"нечетные"=>[1,3,5,7], "четные"=>[2,4,6,8]}
{"foo"=>123, [4,5,6]=>"my array", "867-5309"=>"Jenny"}
```

К содержимому хэша-переменной доступ осуществляется так же, как для массивов, – с помощью квадратных скобок:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
```

Однако следует подчеркнуть, что у массивов и хэшей много методов, именно они и делают эти контейнеры полезными. Ниже, в разделе «ООП в Ruby», мы раскроем эту тему более подробно.

1.2.4. Операторы и приоритеты

Познакомившись с основными типами данных, перейдем к операторам в языке Ruby. В приведенном ниже списке они представлены в порядке убывания приоритета:

`::`	Разрешение области видимости
`[]`	Взятие индекса
`**`	Возведение в степень
`+ - ! ~`	Унарный плюс/минус, НЕ ...
`* / %`	Умножение, деление ...
`+ -`	Сложение/вычитание
`<< >>`	Логические сдвиги ...
`&`	Поразрядное И
`| ^`	Поразрядное ИЛИ, исключающее ИЛИ
`> >= < <=`	Сравнение
`== === <=> != =~ !~`	Равенство, неравенство ...
`&&`	Логическое И
`||`	Логическое ИЛИ
`.. ...`	Операторы диапазона
`= (also +=, -=, ...)`	Присваивание
`?:`	Тернарный выбор
`not`	Логическое отрицание
`and or`	Логическое И, ИЛИ

Некоторые из перечисленных символов служат сразу нескольким целям. Например, оператор `<<` обозначает поразрядный сдвиг влево, но также применяется для добавления в конец (массива, строки и т. д.) и как маркер встроенного документа. Аналогично знак + означает сложение чисел и конкатенацию строк. Ниже мы увидим, что многие операторы – это просто сокращенная запись вызова методов.

Итак, мы определили большую часть типов данных и многие из возможных над ними операций. Прежде чем двигаться дальше, приведем пример программы.

1.2.5. Пример программы

В любом руководстве первой всегда приводят программу, печатающую строку `Hello, world!`, но мы рассмотрим что-нибудь более содержательное. Вот небольшая интерактивная консольная программа, позволяющая переводить температуру из шкалы Фаренгейта в шкалу Цельсия и наоборот.

```
print "Введите температуру и шкалу (C or F): "
str = gets
exit if str.nil? or str.empty?
str.chomp!
temp, scale = str.split(" ")

abort "#{temp} недопустимое число." if temp !~ /-?\d+/

temp = temp.to_f
case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
else
  abort "Необходимо задать C или F."
end

if f.nil?
  print "#{c} градусов C\n"
else
  print "#{f} градусов F\n"
end
```

Ниже приведены примеры прогона этой программы. Показано, как она переводит градусы Фаренгейта в градусы Цельсия и наоборот, а также как обрабатывает неправильно заданную шкалу или число:

```
Введите температуру и шкалу (C or F): 98.6 F
37.0 градусов C

Введите температуру и шкалу (C or F): 100 C
212.0 градусов F

Введите температуру и шкалу (C or F): 92 G
Необходимо задать C или F.

Введите температуру и шкалу (C or F): junk F
junk недопустимое число.
```

Теперь рассмотрим, как эта программа работает. Все начинается с предложения `print`, которое есть не что иное, как вызов метода `print` из модуля `Kernel`. Данный метод выполняет печать на стандартный вывод. Это самый простой способ оставить курсор в конце строки.

Далее мы вызываем метод `gets` (прочитать строку из стандартного ввода) и присваиваем полученное значение переменной `str`. Для удаления хвостового символа новой строки вызывается метод `chomp!`.

Обратите внимание, что `print` и `gets`, которые выглядят как «свободные» функции, на самом деле являются методами класса `Object` (который, вероятно, наследует `Kernel`). Точно так же `chomp!` – метод, вызываемый от имени объекта

`str`. При вызовах методов в Ruby обычно можно опускать скобки: `print "foo"` и `print("foo")` – одно и то же.

В переменной `str` хранится символьная строка, но могли бы храниться данные любого другого типа. В Ruby данные имеют тип, а переменные – нет. Переменная начинает существовать, как только интерпретатор распознает присваивание ей; никаких предварительных объявлений не существует.

Метод `exit` завершает программу. В той же строке мы видим управляющую конструкцию, которая называется «модификатор `if`». Он аналогичен предложению `if`, существующему в большинстве языков, только располагается после действия. Для модификатора `if` нельзя задать ветвь `else`, и он не требует закрытия. Что касается условия, мы проверяем две вещи: имеет ли переменная `str` значение (то есть не равна `nil`) и не является ли она пустой строкой. Если встретится конец файла, то будет истинно первое условие; если же пользователь нажмет клавишу Enter, не введя никаких данных, – второе.

Это предложение можно было бы записать и по-другому:

```
exit if not str or not str[0]
```

Эти проверки работают потому, что переменная может иметь значение `nil`, а `nil` в Ruby в логическом контексте вычисляется как «ложно». На самом деле как «ложно» вычисляются `nil` и `false`, а все остальное – как «истинно». Это означает, кстати, что пустая строка `""` и число 0 – не «ложно».

В следующем предложении над строкой выполняется операция `chomp!` (для удаления хвостового символа новой строки). Восклицательный знак в конце предупреждает, что операция изменяет значение самой строки, а не возвращает новую. Восклицательный знак применяется во многих подобных ситуациях как напоминание программисту о том, что у метода есть побочное действие или что он более «опасен», чем аналогичный метод без восклицательного знака. Так, метод `chomp` возвращает такой же результат, но не модифицирует значение строки, для которой вызван.

В следующем предложении мы видим пример множественного присваивания. Метод `split` разбивает строку на куски по пробелам и возвращает массив. Двум переменным в левой части оператора присваиваются значения первых двух элементов массива в правой части.

В следующем предложении `if` с помощью простого регулярного выражения выясняется, введено ли допустимое число. Если строка не соответствует образцу, который состоит из необязательного знака «минус» и одной или более цифр, то число считается недопустимым и программа завершается. Отметим, что предложение `if` оканчивается ключевым словом `end`. Хотя в данном случае это не нужно, мы могли бы включить перед `end` ветвь `else`. Ключевое слово `then` необязательно; в этой книге мы стараемся не употреблять его.

Метод `to_f` преобразует строку в число с плавающей точкой. Это число записывается в ту же переменную `temp`, в которой раньше хранилась строка.

Предложение `case` выбирает одну из трех ветвей: пользователь указал `C`, `F` или какое-то другое значение в качестве шкалы. В первых двух случаях выполняется вычисление, в третьем мы печатаем сообщение об ошибке и выходим.

Кстати, предложение case в Ruby позволяет гораздо больше, чем показано в примере. Нет никаких ограничений на типы данных, а все выражения могут быть произвольно сложными, в том числе диапазонами или регулярными выражениями.

В самом вычислении нет ничего интересного. Но обратите внимание, что переменные c и f впервые встречаются внутри ветвей case. В Ruby нет никаких объявлений – переменная начинает существовать только в результате присваивания. А это означает, что после выхода из case лишь одна из переменных c и f будет иметь действительное значение.

Мы воспользовались этим фактом, чтобы понять, какая ветвь исполнялась, и в зависимости от этого вывести то или другое сообщение. Сравнение f с nil позволяет узнать, есть ли у переменной осмысленное значение. Этот прием применен только для демонстрации возможности: ясно, что при желании можно было бы поместить печать прямо внутрь предложения case.

Внимательный читатель заметит, что мы пользовались только «локальными» переменными. Это может показаться странным, так как, на первый взгляд, их областью видимости является вся программа. На самом деле они локальны относительно верхнего уровня программы. Глобальными они кажутся лишь потому, что в этой простой программе нет контекстов более низкого уровня. Но если бы мы объявили какие-нибудь классы или методы, то в них переменные верхнего уровня были бы не видны.

1.2.6. Циклы и ветвление

Потратим немного времени на изучение управляющих конструкций. Мы уже видели простое предложение if и модификатор if. Существуют также парные структуры, в которых используется ключевое слово unless (в них также может присутствовать необязательная ветвь else), а равно применяемые в выражениях формы if и unless. Все они сведены в таблицу 1.1.

Таблица 1.1. Условные предложения

Формы с if	Формы с unless

`if x < 5 then`
`  statement1`
`end`
	`unless x >= 5 then`
	`  statement1`
	`end`

`if x < 5 then`
`  statement1`
`else`
`  statement2`
`end`
	`unless x < 5 then`
	`  statement2`
	`else`
	`  statement1`
	`end`

`statement1 if y == 3`	`statement1 unless y != 3`
`x = if a>0 then b else c end`	`x = unless a<=0 then c else b end`

Здесь формы с ключевыми словами if и unless, расположенные в одной строке, выполняют в точности одинаковые функции. Обратите внимание, что слово

then можно опускать во всех случаях, кроме последнего (предназначенного для использования в выражениях). Также заметьте, что в модификаторах (третья строка) ветви else быть не может.

Предложение case в Ruby позволяет больше, чем в других языках. В его ветвях можно проверять различные условия, а не только сравнивать на равенство. Так, например, разрешено сопоставление с регулярным выражением. Проверки в предложении case эквивалентны оператору ветвящегося равенства (===), поведение которого зависит от объекта. Рассмотрим пример:

```
case "Это строка символов."
  when "одно значение"
    puts "Ветвь 1"
  when "другое значение"
    puts "Ветвь 2"
  when /симв/
    puts "Ветвь 3"
  else
    puts "Ветвь 4"
end
```

Этот код напечатает Ветвь 3. Почему? Сначала проверяемое выражение сравнивается на равенство с двумя строками: "одно значение" и "другое значение". Эта проверка завершается неудачно, поэтому мы переходим к третьей ветви. Там находится образец, с которым сопоставляется выражение. Поскольку оно соответствует образцу, то выполняется предложение print. В ветви else обрабатывается случай, когда ни одна из предшествующих проверок не прошла.

Если проверяемое выражение – целое число, то его можно сравнивать с целочисленным диапазоном (например, 3..8); тогда проверяется, что число попадает в диапазон. В любом случае выполняется код в первой подошедшей ветви.

В Ruby имеется богатый набор циклических конструкций. К примеру, while и until – циклы с предварительной проверкой условия, и оба работают привычным образом: в первом случае задается условие продолжения цикла, а во втором – условие завершения. Есть также их формы с модификатором, как для предложений if и unless. Кроме того, в модуле Kernel есть метод loop (по умолчанию бесконечный цикл), а в некоторых классах реализованы итераторы.

В примерах из таблицы 1.2 предполагается, что где-то определен такой массив list:

```
list = %w[alpha bravo charlie delta echo];
```

В цикле этот массив обходится и печатается каждый его элемент.

Таблица 1.2. Циклы

`# Цикл 1 (while)`
`i=0`
`while i < list.size do`
`  print "#{list[i]} "`
`  i += 1`
`end`
	`# Цикл 2 (until)`
	`i=0`
	`until i == list.size do`
	`  print "#{list[i]} "`
	`  i += 1`
	`end`

Таблица 1.2. Циклы

`# Цикл 3 (for)`
`for x in list do`
`  print "#{x} "`
`end`
	`# Цикл 4 (итератор 'each')`
	`list.each do |x|`
	`  print "#{x} "`
	`end`

`# Цикл 5 (метод 'loop')`
`i=0`
`n=list.size-1`
`loop do`
`  print "#{list[i]} "`
`  i += 1`
`  break if i > n`
`end`
	`# Цикл 6 (метод 'loop')`
	`i=0`
	`n=list.size-1`
	`loop do`
	`  print "#{list[i]} "`
	`  i += 1`
	`  break unless i <= n`
	`end`

`# Цикл 7 (итератор 'times')`
`n=list.size`
`n.times do |i|`
`  print "#{list[i]} "`
`end`
	`# Цикл 8 (итератор 'upto')`
	`n=list.size-1`
	`0.upto(n) do |i|`
	`  print "#{list[i]} "`
	`end`

`# Цикл 9 (for)`
`n=list.size-1`
`for`
`  i in 0..n do`
`  print "#{list[i]} "`
`end`
	`# Цикл 10 ('each_index')`
	`list.each_index do |xt`
	`  print "#{list[x]} "`
	`end`

Рассмотрим эти примеры более подробно. Циклы 1 и 2 – «стандартные» формы циклов `while` и `until`; ведут они себя практически одинаково, только условия противоположны. Циклы 3 и 4 – варианты предыдущих с проверкой условия в конце, а не в начале итерации. Отметим, что использование слов `begin` и `end` в этом контексте – просто грязный трюк; на самом деле это был бы блок begin/end (применяемый для обработки исключений), за которым следует модификатор `while` или `until`. Однако для тех, кто желает написать цикл с проверкой в конце, разницы нет.

На мой взгляд, конструкции 3 и 4 – самый «правильный» способ кодирования циклов. Они заметно проще всех остальных: нет ни явной инициализации, ни явной проверки или инкремента. Это возможно потому, что массив «знает» свой размер, а стандартный итератор `each` (цикл 6) обрабатывает такие детали автоматически. На самом деле в цикле 5 производится неявное обращение к этому итератору, поскольку цикл `for` работает с любым объектом, для которого определен итератор `each`. Цикл `for` – лишь сокращенная запись для вызова `each`; часто такие сокращения называют «синтаксической глазурью», имея в виду, что это не более чем удобная альтернативная форма другой синтаксической конструкции.

В циклах 5 и 6 используется конструкция `loop`. Выше мы уже отмечали, что хотя `loop` выглядит как ключевое слово, на самом деле это метод модуля `Kernel`, а вовсе не управляющая конструкция.