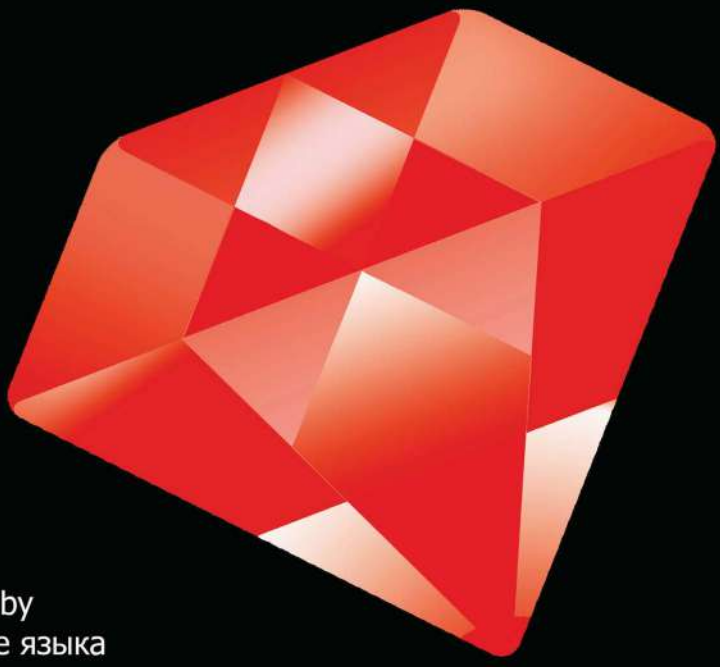
самоучитель

Игорь Симдянов

Ruby



Новинки языка Ruby
Полное изложение языка
Веб-программирование
Автоматическое тестирование
150 заданий

Материалы
на www.bhv.ru

bhv®

Игорь Симдянов

самоучитель Ruby

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.438 Ruby
ББК 32.973.26-018.1
С37

Симдянов И. В.

С37 Самоучитель Ruby. — СПб.: БХВ-Петербург, 2020. — 656 с.: ил. — (Самоучитель)

ISBN 978-5-9775-4060-5

Язык Ruby излагается последовательно от простого к сложному. Описываются интерпретатор Ruby, утилиты, детально рассматривается современная Ruby-экосистема, работа со стандартной и сторонними библиотеками Дан разбор синтаксических конструкций: операторов, переменных, констант, конструкций ветвления и циклов, блоков и итераторов. Подробно описаны объектно-ориентированные возможности Ruby: классы, модули, объекты и методы. Показано практическое применение языка Ruby в веб-программировании и автоматическом тестировании. Для закрепления материала в конце глав приводятся задания. С помощью книги можно не только освоить язык Ruby, но и подготовиться к работе с профессиональными фреймворками: Ruby on Rails, Sinatra, RSpec, MiniTest и Cucumber. Опытных разработчиков может заинтересовать подробное описание нововведений версий от 2.0 до 2.6. Электронный архив с исходными кодами доступен на сайте издательства и GitHub.

Для программистов

УДК 004.438 Ruby
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Карины Соловьевой*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-4060-5

Оглавление

Предисловие

Цель книги

Цель книги — полное и последовательное изложение языка программирования Ruby. Книга рассчитана как на начинающих разработчиков, не владеющих ни одним языком программирования, так и на опытных программистов, желающих освоить Ruby. Ruby-разработчикам со стажем книга также будет полезна, поскольку освещает нововведения языка, начиная с версии 2.0 до версии 2.6.

При создании книги не ставилась задача создать иллюзию, будто язык Ruby простой и не потребует усилий для его освоения. Ruby не является C-подобным и часто использует уникальные конструкции и решения, не имеющие аналогов в других живых языках программирования. Казалось бы, знакомые по другим языкам конструкции часто ведут себя в нем немного по-другому или в корне имеют иное назначение и свои особенности синтаксиса.

Некоторые главы могут показаться сложными даже опытным разработчикам и потребуют минимум двукратного прочтения. Кроме того, не следует пренебрегать экспериментами в консоли, решением задач и использованием полученных из книги знаний на практике. Любая практическая работа с кодом здорово ускоряет усвоение и запоминание материала.

Как создавалась книга...

Книга была написана за 6 месяцев, однако подготовка к ее созданию заняла 7 лет, на протяжении которых я работал Ruby-разработчиком в компаниях Newsmedia, Rambler&Co и Mail.ru. В каждой из них вел авторские курсы, посвященные программированию баз данных, программированию на Ruby, автоматическому тестированию и веб-программированию с использованием фреймворка Ruby on Rails. Однако большая часть времени была посвящена разработке Ruby-проектов: портал Rambler.ru, телеканал Life, газета izvestia.ru.

Терминология

Влияние западной IT-индустрии необычайно велико. В Российской Федерации фактически вся аппаратная часть, программное обеспечение, языки программирования и технологии заимствуются с Запада. Несмотря на то, что для многих терминов можно подобрать подходящие альтернативы, речь разработчиков изобилует профессиональным сленгом.

В большинстве случаев связано это с тем, что специалисты все чаще и чаще прибегают к литературе и видеоматериалам на английском языке, которых больше и которые выигрывают в актуальности.

Такая ситуация характерна не только для Российской Федерации — с похожими явлениями сталкиваются почти все страны. Ряд специалистов резко выступают против такого засорения их собственного языка. Однако мы будем рассматривать эту ситуацию как расширение и обогащение его новыми терминами. Профессиональные сленговые выражения вполне прижились в российском IT-сообществе и в русском языке.

Чтобы понимать коллег и говорить с ними на одном языке, сленг необходимо понимать. Терминология Ruby-сообщества весьма уникальна даже по меркам западной индустрии. Например, Ruby-библиотеки называются *гемами* (gems). Здесь имеет место своеобразная игра слов: «ruby» переводится с английского как «рубин», а «gems» — как «драгоценные камни». В такой ситуации переводить gem на русский язык — идея сомнительная. Обозначать гемы более привычным для российского уха термином *библиотека* (library) тоже не совсем точно. Тем более, что в русском Ruby-сообществе термин *гем* прижился и интенсивно используется.

Ruby — абсолютно объектно-ориентированный язык, и по сравнению с традиционными языками у него имеется довольно много контекстов определения и вызова методов. Поэтому различают *методы классов*, *инстанс-методы*, *синглетон-методы*. Для первых двух терминов можно использовать названия *методы классов* и *методы объектов*, однако для синглетон-методов пришлось бы вводить что-то вроде «метод единичного объекта». Длинные неуклюжие термины всегда проигрывают коротким и емким. Поэтому на практике рубисты продолжают использовать понятие синглетон-метода, чтобы ни писали в книгах.

Язык — это стихия, и какие бы иллюзии люди ни испытывали на предмет того, что они управляют состоянием языка, развивается он все же по своим законам. И в этой книге мы не станем бороться с терминологией, а будем использовать ее в том виде, в котором она сложилась в российском Ruby-сообществе.

Исходные коды

Исходные коды, приведенные в книге, можно найти в сопровождающем ее электронном архиве (см. *приложение 2*). В начале каждой главы указывается каталог архива, в котором содержатся примеры этой главы, а в заголовках листингов приводятся названия соответствующих файлов.

Сам электронный архив к книге выложен на FTP-сервер издательства «БХВ-Петербург» по адресу: **ftp://ftp.bhv.ru/9785977540605.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru** (см. *приложение 2*).

Исходные коды к книге также размещены на GitHub-аккаунте по адресу: **https://github.com/igorsimdyanov/ruby**.

По ссылке: **https://github.com/igorsimdyanov/ruby/issues** или по почтовому адресу издательства: **mail@bhv.ru** читатели могут адресовать свои вопросы автору.

Задания

Каждая глава снабжается заданиями, которые побуждают исследовать документацию, знакомиться с не вошедшими в книгу методами и гемами, читать статьи, искать решение. Всего книга содержит 150 таких заданий.

Типографские соглашения

В книге приняты следующие соглашения:

- *курсив* применяется для выделения терминов, когда они употребляются впервые, им дается определение или раскрывается их природа;
- **полужирный шрифт** служит для выделения интернет-адресов (URL);
- моноширинный шрифт используется для представления содержимого листингов, имен переменных, констант, классов и модулей, а также для выделения команд и названий утилит;
- названия файлов и каталогов традиционно для издательства «БХВ-Петербург» выделяются шрифтом Arial.

Приведенные в книге примеры кода — в зависимости от среды выполнения — оформляются по-разному. Так, при запуске команды в консоли перед ней указывается символ доллара: `$`. Этот символ используется в качестве приглашения в командных оболочках UNIX-подобных операционных систем, однако подавляющее большинство команд работают точно так же и в операционной системе Windows. Места в книге, где команда явно зависит от типа операционной системы, каждый раз отмечаются явно.

Однако обратите внимание: при наборе команды для выполнения примера символ `$` вводить не следует — он служит лишь для обозначения среды, в которой выполняется команда.

Текст, который вводится пользователем, выделяется полужирным шрифтом, результат вывода команды — обычным:

```
$ erb template.erb
Выражение 2 + 2 = 4
```

При выполнении Ruby-кода в среде интерактивного Ruby `irb` строка ввода предваряется символом `>`, а ответ среды — последовательностью `=>`. Для того чтобы вы-

полнить такой код, потребуется запустить утилиту `irb` (см. *главу 3*). При этом пользовательский ввод так же выделяется полужирным:

```
 > 'Hello, world!'.size
=> 13
```

В том случае, когда пример расположен в отдельном файле, его можно будет найти в электронном архиве, сопровождающем книгу. Код таких файлов предваряется «шапкой» с номером листинга и ссылкой на файл:

Листинг 4.14. Создание символа. Файл symbol.rb

```
p :white
```

Иногда результат работы программы выводится в виде Ruby-комментария, который начинается с символа `#`:

Листинг 4.2. Сложение чисел. Файл sum.rb

```
puts 2 + 2 # 4
```

В начале каждой главы, как уже отмечалось ранее, указывается название каталога электронного архива, в котором содержатся файлы примеров, приведенных в этой главе.

Благодарности

Программиста редко можно заставить писать связные комментарии, не говоря уже о техническом тексте. Пробравшись через барьеры языка, технологий, отладки кода, они настолько погружаются в мир кодирования, что вытащить их из него и заинтересовать чем-то другим не представляется возможным.

Излагать свои мысли, «видеть» текст — это, скорее, не искусство или особый дар, а навык, который необходимо осваивать. В научном мире это такой же рядовой инструмент, как компьютер, владеть им должен каждый, не зависимо от способностей. Я очень благодарен Зеленцову Сергею Васильевичу, моему научному руководителю, который потратил безумное количество времени в попытках научить меня писать.

Второй человек, благодаря которому вы смогли увидеть эту и множество других книг, это Максим Кузнецов: наука, война, медицина, психология, химия, физика музыка, программирование — для него не было запретных областей. Он везде был свой и чувствовал себя как рыба в воде. Он быстро жил, и жизнь его быстро закончилась. Остались спасенные им люди. Эта книга, в том числе, и его детище.

С языком программирования Ruby я познакомился в компании Newsmedia, перерабатывая легаси-проекты на PHP в современные Ruby-приложения. Деловая и дружелюбная атмосфера в команде, семинары и школа программирования напоминали

мне атмосферу научной лаборатории, в которой началась моя карьера программиста.

Материал этой книги был отточен и отчитан на курсах Geekbrains (компания Mail.ru). Благодаря бурному развитию компании и поиску новых форматов, мне удалось перепробовать себя во всех жанрах: преподаватель, методист, автор видеокурсов, рецензент, наставник. Благодаря вопросам, которые задавали ученики, мне удалось погрузиться в атмосферу, которая окружает начинающего рубиста. Именно на курсах окончательно сформировался план и материал этой книги.

В настоящий момент я работаю в компании Rambler&Co (rambler.ru, gazeta.ru, lenta.ru, championat.com, okko.tv, livejournal.com и т. д.). Благодаря Rambler&Co мне посчастливилось участвовать в крупных высоконагруженных проектах и познакомиться со многими удивительными и талантливыми людьми.

Выразить благодарность хотелось бы всем разработчикам и системным администраторам, с которыми мне довелось поработать в проектах. Каждый из них повлиял на мой стиль программирования и на содержимое этой книги: Евгений Аббакумов, Алексей Авдеев, Олег Афанасьев, Семен Багреев, Александр Бобков, Ян Бодетский, Игорь Варавко, Михаил Волков, Артемий Гаврелюк, Стас Герман, Никита Головин, Александр Горбунов, Александр Григоров, Вадим Жарко, Сергей Зубков, Михаил Кобзев, Николай Коваленко, Илья Конюхов, Александр Краснощеков, Денис Максимов, Алексей Мартынюк, Михаил Милюткин, Дмитрий Михеев, Александр Муравкин, Александр Никифоренко, Артем Нистратов, Вячеслав Онуфриев, Павел Петлинский, Алексей Похожаев, Сергей Пузырев, Вячеслав Савельев, Сергей Савостьянов, Андрей Синтинков, Сергей Суматохин, Екатерина Фонова, Владимир Чиликов.

Кроме того, хотел бы выразить благодарность коллективу издательства «БХВ-Петербург», в особенности Евгению Рыбакову и Григорию Добину, без которых эта книга не была бы издана.

ГЛАВА 1

Введение в язык Ruby

Файлы с исходными кодами этой главы находятся в каталоге *intro* сопровождающего книгу электронного архива.

В первой главе мы познакомимся с языком программирования Ruby, его достоинствами и недостатками, историей развития, а также с нишей, которую этот язык занимает в современном компьютерном мире. Кроме того, установим Ruby и попробуем воспользоваться интерпретатором для запуска первой программы.

1.1. Философия Ruby

Ruby создан японским программистом Юкихиро Мацумото в 1995 году. С английского языка Ruby переводится как «рубин», а произносится «руби». Название родилось под влиянием языка Perl (что созвучно английскому pearl, «жемчужина»).

С точки зрения Юкихиро Мацумото, все существующие на тот момент языки были недостаточно «объектно-ориентированные», поэтому он создал свой собственный. Ему удалось создать элегантный и удобный язык, который приобрел популярность сначала в Японии, а после перевода документации в 1997 году на английский язык и во всем остальном мире.

Характеризуя язык, Мацумото описывает его возникновение следующими словами:

> *До создания Ruby я изучил множество языков, но никогда не испытывал от них полного удовлетворения. Они были уродливее, труднее, сложнее или проще, чем я ожидал. И мне захотелось создать свой собственный язык, который смог бы удовлетворить мои программистские запросы. О целевой аудитории, для которой предназначался язык, я знал вполне достаточно, поскольку сам был ее представителем. К моему удивлению, множество программистов по всему миру испытали чувства, сходные с моими. Открывая для себя Ruby и программируя на нем, они получают удовольствие.*
>
> *Разрабатывая язык Ruby, я направил всю свою энергию на ускорение и упрощение процесса программирования. Все свойства Ruby, включая объектную ориентацию, сконструированы так, чтобы при своей работе они оправдывали ожидания средних по классу программистов (например, мои собственные). Большинство*

программистов считают этот язык элегантным, легкодоступным и приятным для программирования.

В отличие от многих существующих на тот момент языков программирования, Ruby изначально проектировался как полностью объектно-ориентированный язык без базовых примитивных типов. Буквально любая конструкция языка, за исключением горстки ключевых слов, является либо объектом, либо методом какого-либо объекта.

В Ruby весьма много соглашений и сокращений, на которых мы подробно будем останавливаться на протяжении всей книги. Эти соглашения вкупе с элегантным синтаксисом приводят к довольно-таки компактному коду. В результате сложный проект можно разработать силами небольшой команды.

Программы разрабатывают люди, которым свойственно ошибаться. Чем больше кодовая база, тем больше времени необходимо потратить на ее обслуживание: внесение изменений требует много времени на анализ существующего кода, и чем объемнее код, тем больше в нем ошибок. Поэтому в Ruby поощряется минималистичный подход — если что-то можно сократить, код обязательно подвергается сокращению. На протяжении книги мы многократно в этом убедимся.

Ruby — интерпретируемый динамический язык с необычно развитыми средствами метапрограммирования. Последние позволяют весьма сильно преобразовывать язык, менять поведение не только разрабатываемых, но и уже существующих объектов и классов.

Если в других языках программирования метапрограммирование либо не развито, либо находится под негласным запретом, в Ruby оно крайне популярно, что приводит к созданию многочисленных DSL-языков (domain-specific language, предметно-ориентированный язык).

В 2004 году Дэвид Ханссон с использованием Ruby разработал веб-фреймворк Ruby on Rails, который предназначался для быстрой разработки веб-сайтов. *Фреймворк* — это набор библиотек и утилит для генерации кода, которые позволяют разрабатывать приложение не с нуля, а начиная с заготовки. Ruby-сообщество получило великолепный инструмент, которому, благодаря позднему выходу на рынок (к этому времени сайты разрабатывались на Perl, PHP, .NET), удалось избежать многих архитектурных ошибок, совершенных в конкурирующих языках. Если в PHP и JavaScript сложилось множество несовместимых друг с другом фреймворков, усилия Ruby-сообщества были сосредоточены фактически на одном, который объединил в единую экосистему остальные Ruby-фреймворки. Ruby on Rails на самом деле представляет собой множество фреймворков, каждый из которых может быть заменен альтернативной реализацией, как в конструкторе. С одной стороны, это позволяет добиться высокой гибкости системы, с другой — быстро впитывать удачные наработки, которые появляются внутри сообщества.

По сегодняшний день Ruby on Rails служит источником вдохновения и идей, которые реализуются в альтернативных веб-фреймворках: Django на Python, Laravel в PHP, Spring в Java.

Именно благодаря фреймворку Ruby on Rails, язык Ruby получил большую популярность в мире. Если на родине языка, в Японии, Ruby часто применяется в разных сферах: от встроенных решений до программирования промышленных роботов, то во всем остальном мире Ruby используется главным образом в веб-программировании. Впрочем, красота и элегантность Ruby привлекает не только веб-разработчиков.

Еще одна традиционно сильная сторона языка Ruby — автоматическое тестирование. Благодаря тому, что на Ruby очень легко создаются декларативные DSL-языки, существует большое количество Ruby-фреймворков для тестирования программного обеспечения: MiniTest, RSpec, Cucumber. Последний фреймворк, позволяющий создать приемочные тесты и описать спецификацию программного обеспечения, широко известен за пределами Ruby-сообщества.

Ruby используется для создания утилит, выполняющих доставку программного обеспечения на серверы: Pupet, Chef, Capistrano. Кроме того, он задействован в системе конфигурирования виртуальных машин Vagrant.

Ruby — не самый быстрый язык по скорости выполнения, однако определенно это один из самых быстрых языков по скорости разработки. Именно поэтому он популярен в стартапах: небольшой компактной командой можно быстро разработать масштабируемое решение и захватить рынок. Получив прибыль, можно перерабатывать систему и переписывать на другие языки программирования. Именно таким образом состоялась социальная сеть Twitter, изначально написанная на Ruby. Многие же стартапы так и остаются верными Ruby — например, GitHub.

Язык Ruby, в отличие от многих других языков, не C-подобен, в нем отсутствуют типы, он абсолютно объектно-ориентированный и этим очень походит на Python 3. Функциональные возможности языка Ruby играют ключевую роль. Несмотря на наличие классических конструкций циклов `for`, `while` и `until`, в Ruby-сообществе присутствует негласный запрет на их использование, — вместо них в подавляющем большинстве случаев задействуются блоки и итераторы.

Таким образом, в Ruby многие знакомые из других языков конструкции имеют уникальные поведение и свойства. Начать кодировать на Ruby легко, однако для получения эффективного и элегантного кода потребуется затратить некоторое время на изучение как самого языка, так и его экосистемы.

1.2. Реализации Ruby

В компилируемых языках программирования, таких как C или C++, результатом сборки программы является исполняемый код, т. е. набор машинных инструкций, которые центральный процессор может выполнять без предварительной подготовки.

В интерпретируемых языках программирования, к которым относится Ruby, программа выполняется другой программой — *интерпретатором*. Ее необходимо устанавливать везде, где требуется выполнение Ruby-кода.

Существует несколько реализаций интерпретатора Ruby, и одна из наиболее известных — MRI (Matsumoto Ruby Interpreter), которую развивает и поддерживает создатель языка Юкихиро Мацумото. Эта реализация де-факто является стандартом языка, все остальные реализации догоняют эту версию или немного искажают.

Второй по популярности реализацией является JRuby — реализация Ruby в среде виртуальной машины Java. Популярность этой реализации основывается на более высокой скорости по сравнению с MRI за счет большего потребления оперативной памяти.

В операционных системах программный код выполняется в виде параллельных процессов. В рамках каждого из процессов может выполняться один или несколько параллельных потоков (рис. 1.1).

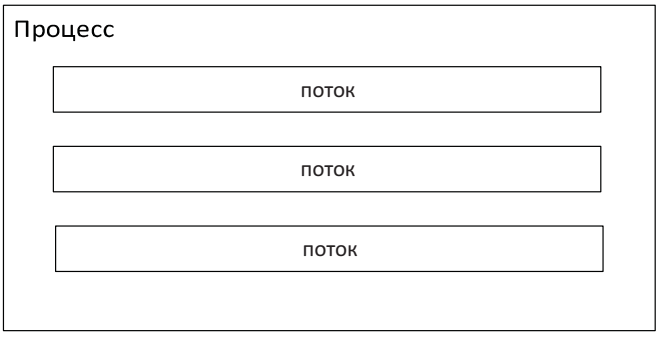


Рис. 1.1. Процессы могут содержать несколько потоков

Все современные интерпретируемые языки разрабатывались во времена, когда центральный процессор содержал только одно ядро. В результате реализация потоков в Ruby, Python, PHP основывается на глобальной блокировке интерпретатора (GIL, Global Interpreter Lock), благодаря чему из нескольких потоков в каждый момент времени может выполняться только один, сколько бы ядер ни предоставлял процессор (рис. 1.2).

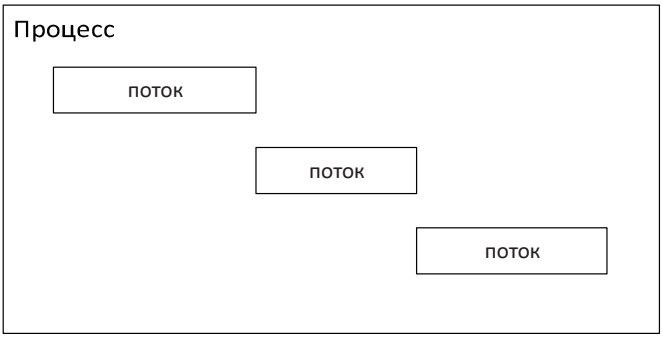


Рис. 1.2. В случае GIL в каждый момент времени может обновляться лишь один поток

В виртуальной машине Java реализованы полноценные потоки — и, как следствие, JRuby получает к ним доступ.

> ***Замечание***
>
> Впрочем, трудности параллельного выполнения нескольких потоков обходятся запуском нескольких параллельных процессов или организацией неблокирующего опроса задач в рамках одного потока (паттерн EventLoop). Эти приемы с успехом используются для создания веб-серверов на Ruby.

Существуют и более экзотические реализации Ruby-интерпретатора:

- Rubinius — реализация Ruby на Ruby;
- MacRuby — реализация Ruby на Objective-C (используется в macOS);
- IronRuby — реализация Ruby для платформы .NET.

Еще раз подчеркнем, что самой распространенной является MRI-реализация — именно ее мы будем использовать на протяжении книги.

1.3. Версии

Версия современного программного обеспечения обычно состоит из трех цифр — например, 2.5.3. Первая цифра называется *мажорной*, вторая — *минорной*, а последняя *патч-версией* (рис. 1.3).

- Мажорная версия изменяется редко, между этими событиями проходят годы, иногда десятилетия. Изменением мажорной версии, как правило, отмечаются существенные архитектурные преобразования в языке, не редко с нарушением обратной совместимости с предыдущими версиями.
- Минорной версией отмечается релиз, в рамках которого добавляется новый функционал (методы, операторы) и исправляются найденные ошибки.
- В рамках патч-версий изменения в синтаксисе и возможностях языка не производятся. Как правило, исправляются критические уязвимости и ошибки.

Уточнить последнюю актуальную версию Ruby для MRI-реализации можно из новостей официального сайта: **https://www.ruby-lang.org/en/news/**.

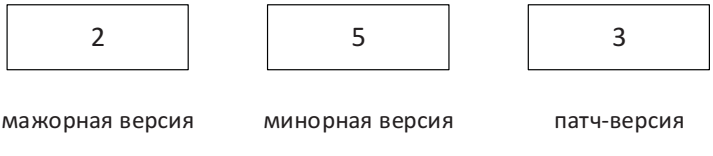


Рис. 1.3. Версионирование Ruby

1.4. Установка Ruby

В некоторых операционных системах Ruby установлен по умолчанию (macOS, ряд дистрибутивов Linux). Впрочем, в этом случае версия Ruby не всегда актуальная, что может быть весьма критично, особенно при веб-разработке, чувствительной к версии из-за большого количества зависимостей.

1.4.1. Установка Ruby в Windows

Для установки проще всего воспользоваться мастером установки RubyInstaller, загрузить который можно с сайта: **https://rubyinstaller.org/downloads/**.

В зависимости от разрядности вашей операционной системы, следует выбрать *x*86 вариант дистрибутива — для 32-разрядной операционной системы или *x*64 — для 64-разрядной.

После завершения скачивания в папке загрузок вашего компьютера должен появиться файл вида rubyinstaller-2.5.3-1-x64. Щелкните на этом файле двойным щелчком для запуска процедуры установки.

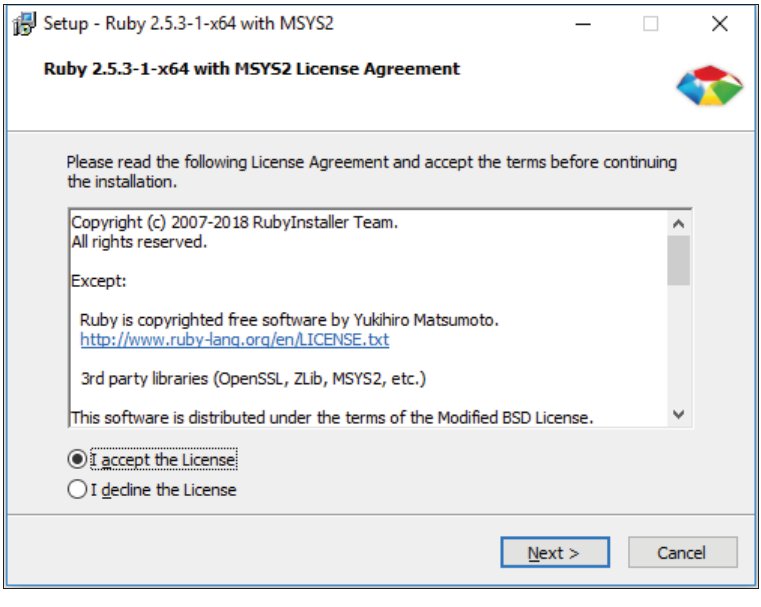


Рис. 1.4. Окно мастера установки RubyInstaller

В первом открывшемся диалоговом окне (рис. 1.4) согласитесь с лицензионным соглашением, выбрав пункт **I accept the License**, и нажмите кнопку **Next** — чтобы запустить процедуру установки (рис. 1.5).

После завершения установки Ruby вам будет предложено установить пакет MSYS2 — введите цифру 3 и нажмите клавишу <Enter> (рис. 1.6). После установки из диалогового окна можно выйти, повторно нажав клавишу <Enter>.

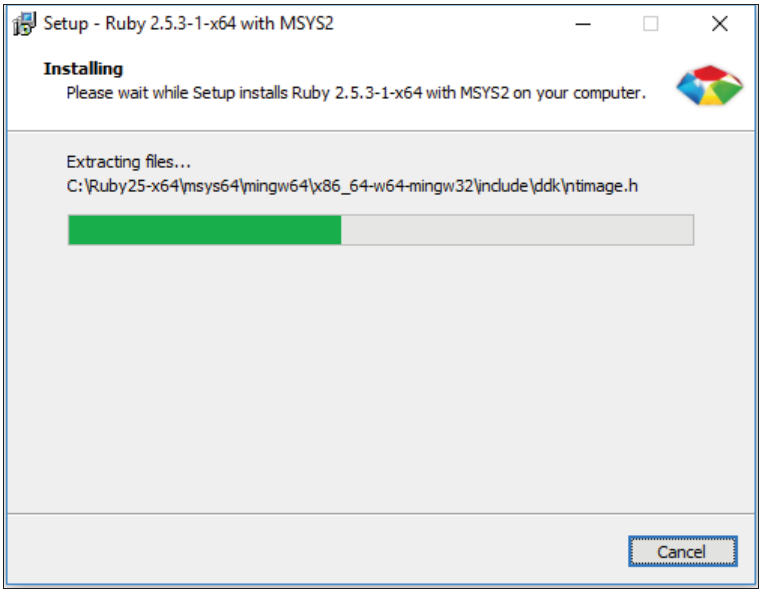


Рис. 1.5. Установка Ruby в Windows

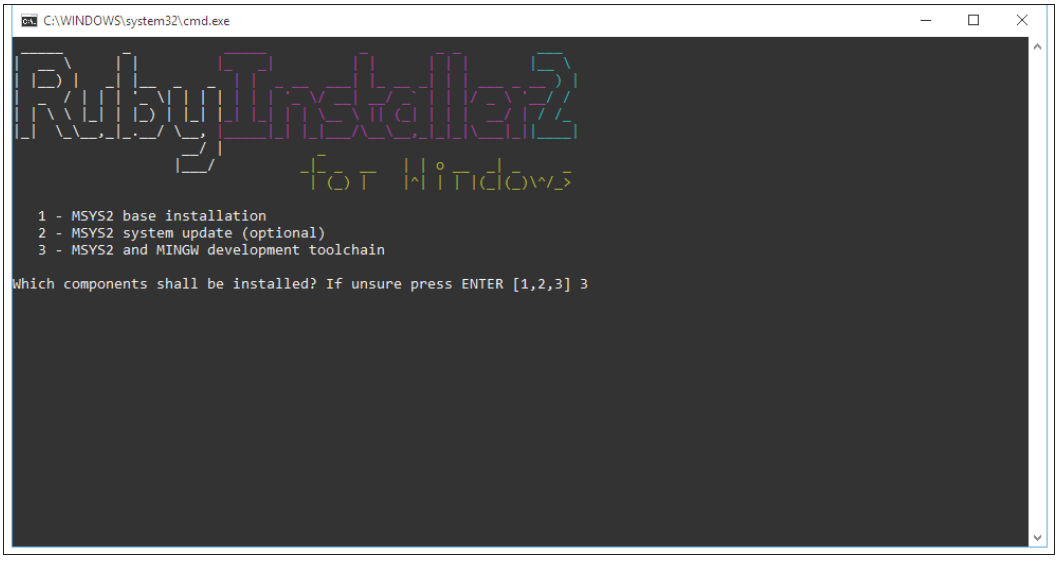


Рис. 1.6. Установка пакета MSYS2

После завершения установки нажмите кнопку **Пуск** и в списке установленных программ найдите папку **Ruby-2.5.3.1** (версия будет отличаться в зависимости от выбранного дистрибутива). Запустите из нее программу **Start Command Promt with Ruby**. Убедитесь, что Ruby успешно установлен, запросив версию языка при помощи команды: `ruby -v`.

1.4.2. Установка Ruby в Linux (Ubuntu)

В UNIX-подобных операционных системах для установки Ruby принято использовать один из менеджеров версий: RVM или rbenv. Преимущество такого подхода заключается в том, что на одном компьютере может быть установлено множество различных версий Ruby, между которыми можно переключаться при помощи менеджера.

Крупный проект нередко содержит множество сторонних библиотек, которые могут использовать возможности языка, появившиеся, начиная с определенной версии. Поэтому, чем крупнее проект, тем он острее зависит от версии Ruby. Чтобы иметь возможность работать одновременно с несколькими проектами, которым требуются разные версии Ruby, необходим один из менеджеров версий.

Кроме того, возможность быстрого переключения с одной версии Ruby на другую упрощает процесс миграции приложения на более современную версию Ruby.

1.4.2.1. Менеджер версий RVM

Менеджер версий RVM (Ruby Version Manager) исторически появился одним из первых. Для его установки следует воспользоваться инструкциями с официального сайта **http://rvm.io** и выполнить по очереди команды из листинга 1.1.

> ***Замечание***
>
> Рекомендуется устанавливать RVM из-под текущего пользователя, а не суперпользователя (root). Все последующие операции с утилитой `rvm` так же выполняются из-под обычного пользователя, без использования команды `sudo`. Это сильно упростит работу со сторонними библиотеками.

Листинг 1.1. Установка RVM. Файл rvm.sh

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-keys
409B6B1796C275462A1703113804BB82D39DC0E3
7D2BAF1CF37B13E2069D6956105BD0E739499BDB
$ sudo apt-get install curl
$ curl -sSL https://get.rvm.io | bash
$ source ~/.rvm/scripts/rvm
$ rvm install 2.6
```

Первая инструкция устанавливает в системе публичный ключ разработчиков — это необходимо для успешной установки менеджера в системе. Вторая строка устанавливает консольную утилиту `curl`, чтобы иметь возможность загрузить RVM по сети. Далее при помощи `curl` скачивается пакет с дистрибутивом и производится его установка.

Установщик автоматически добавит в скрипт ~/.bash_profile строку инициализации:

```
[[ -s "$HOME/.rvm/scripts/rvm" ]] && source "$HOME/.rvm/scripts/rvm"
```

Однако для инициализации RVM необходимо перезапустить консоль. Чтобы этого не делать, можно воспользоваться командой `source`.

Последняя строка из листинга 1.1 осуществляет установку Ruby. Если для вашей операционной системы обнаружен предкомпилированный дистрибутив, RVM загрузит его по сети, в противном случае будет выполнена компиляция программы из исходного кода.

Для того чтобы запросить список всех известных RVM версий Ruby, следует воспользоваться командой `rvm list known` (листинг 1.2).

Листинг 1.2. Получение списка доступных версий Ruby

```
$ rvm list known
# MRI Rubies
[ruby-]2.1[.10]
[ruby-]2.2[.10]
[ruby-]2.3[.8]
[ruby-]2.4[.5]
[ruby-]2.5[.3]
[ruby-]2.6[.0]
...
# IronRuby
ironruby[-1.1.3]
```

Для установки выбранной версии достаточно воспользоваться любым именем из списка, убрав из названия квадратные скобки:

```
$ rvm install ruby-2.6.0
```

Впрочем, уточнения версий, приведенные в квадратных скобках, можно опускать и использовать краткое обозначение версии:

```
$ rvm install 2.6
```

Для того чтобы список версий, который возвращает команда `rvm list known`, оставался актуальным, RVM необходимо регулярно обновлять при помощи следующей команды:

```
$ rvm get head
```

Получить список установленных версий можно, передав утилите `rvm` команду `list` (листинг 1.3).

Листинг 1.3. Список установленных версий Ruby-интерпретатора

```
$ rvm list
=* ruby-2.3.5 [ x86_64 ]
   ruby-2.4.2 [ x86_64 ]
   ruby-2.5.3 [ x86_64 ]
   ruby-2.6.0 [ x86_64 ]
```

Перед списком может располагаться несколько маркеров:

- `*` — версия по умолчанию;
- `=>` — текущая версия;
- `=*` — текущая версия и версия по умолчанию совпадают.

Переключиться на выбранную версию можно, передав утилите `rvm` команду `use` и название версии:

```
$ rvm use 2.6.0
```

Для того чтобы выбранная версия стала версией по умолчанию, в приведенную только что команду необходимо добавить параметр `--default`:

```
$ rvm --default use 2.6.0
```

Убедиться в успешной установке Ruby можно, запросив версию Ruby-интерпретатора с передачей утилите `ruby` параметра `-v`:

```
$ ruby -v
ruby 2.6.0p0 (2018-12-25 revision 66547) [x86_64-darwin15]
```

1.4.2.2. Менеджер версий rbenv

Менеджер rbenv в настоящий момент — это наиболее популярный способ установки Ruby в UNIX-подобных операционных системах. В отличие от RVM, он не подменяет команды оболочки — такие как `cd`, и его работа считается более прозрачной с точки зрения эксплуатации. Поэтому этот менеджер более популярен среди системных администраторов и в условиях продакшен-эксплуатации.

Детально описание менеджера можно обнаружить на GitHub-странице проекта по адресу: **https://github.com/rbenv/rbenv**.

В листинге 1.4 приводятся команды для установки Ruby при помощи менеджера пакетов rbenv.

> ***Замечание***
>
> Так же, как и RVM, рекомендуется устанавливать менеджер rbenv из-под текущего пользователя, а не суперпользователя (root).

Листинг 1.4. Установка rbenv. Файл rbenv.sh

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
$ cd ~/.rbenv && src/configure && make -C src
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ ~/.rbenv/bin/rbenv init
$ curl -fsSL https://github.com/rbenv/rbenv-installer/raw/master/bin/
                                                    rbenv-doctor | bash
$ mkdir -p "$(rbenv root)"/plugins
$ git clone https://github.com/rbenv/ruby-build.git "$(rbenv root)"/plugins/
                                                          ruby-build
$ rbenv install 2.6.1
```

Для того чтобы получить список доступных версий, необходимо воспользоваться командой:

```
$ rbenv install -l
```

Получив список, можно установить выбранную версию, передав ее команде `rbenv install`:

```
$ rbenv install 2.6
```

Список доступных версий можно получить, передав утилите `rbenv` команду `versions`:

```
$ rbenv versions
```

Для переключения версии Ruby используется команда `rbenv local`:

```
$ rbenv local 2.6.0
```

Чтобы выбранная версия Ruby-интерпретатора стала версией по умолчанию и сохранилась при последующих сеансах командной оболочки, необходим заменить команду `local` на `global`:

```
$ rbenv global 2.6.0
```

Убедиться в успешной установке Ruby, можно запросив версию Ruby-интерпретатора с передачей утилите `ruby` параметра `-v`:

```
$ ruby -v
ruby 2.6.0p0 (2018-12-25 revision 66547) [x86_64-darwin15]
```

1.4.3. Установка Ruby в macOS

Операционная система macOS — это коммерческий вариант UNIX, ведущий свою родословную от BSD UNIX (так же являющемся предком FreeBSD). Поэтому все программное обеспечение для UNIX-подобных операционных систем доступно для macOS либо сразу, либо после небольшой адаптации.

Жизнь программиста в операционной системе macOS значительно упрощается, если воспользоваться менеджером пакетов Homebrew. Достаточно его установить, и можно использовать все пакеты, доступные в Linux, — например, менеджеры версий RVM и rbenv (см. *разд. 1.4.2*).

Однако, прежде чем устанавливать Homebrew, следует установить Command Line Tools for XCode из магазина AppStore. XCode — это интегрированная среда разработки приложений для macOS и iOS. Полная загрузка XCode не обязательна — достаточно установить инструменты командной строки и компилятор. Убедиться в том, установлен ли XCode, можно при помощи команды:

```
$ xcode-select -p
/Applications/Xcode.app/Contents/Developer
```

Если вместо указанного в приведенном выводе пути выводится предложение установить Command Line Tools, следует установить этот пакет, выполнив команду:

```
$ xcode-select --install
```

На момент подготовки этой книги установить Homebrew можно было при помощи команды:

```
$ ruby -e "$(curl -fsSL
↳ https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Впрочем, точную команду всегда можно выяснить на официальном сайте **http://brew.sh**. После установки в командной строке будет доступна команда `brew`, при помощи которой можно загружать, удалять и обновлять пакеты с программным обеспечением.

> ***Замечание***
>
> Обратите внимание, что для установки Homebrew используется macruby, который поставляется как часть macOS. Если это ваше первое знакомство с языком, и вам не требуются последние синтаксические нововведения, для изучения материала книги можно воспользоваться встроенным Ruby-интерпретатором macOS.

Теперь можно установить RVM (см. *разд. 1.4.2.1*):

```
$ brew install rvm
```

Или rbenv (см. *разд. 1.4.2.2*):

```
$ brew install rbenv
```

Остальные инструкции по установке и переключению версий аналогичны тем, что приводились ранее в разделе, посвященном установке Ruby в Linux (см. *разд. 1.4.2*).

1.5. Запуск программы на выполнение

Программы на Ruby — это обычные текстовые файлы с расширением rb (сокращение от ruby). Для набора текста программ можно воспользоваться любым редактором: блокнотом, консольным Vim, Sublime Text или специализированной интеграционной средой, специально разработанной под Ruby-проекты, — например, RubyMine.

Давайте разработаем первую программу, в которой выведем фразу `'Hello, world!'`, с которой традиционно начинают изучение языков программирования (листинг 1.5).

> ***Замечание***
>
> В современных языках программирования при разработке программы в конце ее файла принято оставлять пустую строку. В случае автоматической генерации кода это позволяет избежать размещения нескольких инструкций на одной строке. В тексте книги мы будем опускать эту последнюю строку, т. к. она не несет дополнительной информации, однако в коде программ ее рекомендуется добавлять.

Листинг 1.5. Вывод фразы 'Hello, world!'. Файл hello.rb

```
puts 'Hello, world!'
```

Для запуска программы на выполнение достаточно передать название файла hello.rb утилите `ruby`. В ответ интерпретатор выведет фразу **Hello, world!**:

```
$ ruby hello.rb
Hello, world!
```

В UNIX-подобных операционных системах имеется возможность назначить файл исполняемым (см. *главу 28*). Для этого можно воспользоваться командой `chmod`:

```
$ chmod 0755 hello.rb
```

В метаданных программы отмечается, что она выполнится как обычный скрипт:

```
$ ./hello.rb
```

Чтобы иметь возможность запускать Ruby-программы таким образом, в начало программы следует поместить *ши-бенг* (shebang) — специальную строку-комментарий, которая поможет командной оболочке найти интерпретатор Ruby (листинг 1.6).

Листинг 1.6. Использование ши-бенга. Файл shebang.rb

```
#!/usr/bin/env ruby
puts 'Hello, world!'
```

Интерпретатор воспринимает первую строку как комментарий (см. *разд. 2.2*) и игнорирует, в то время как командная оболочка использует ее для поиска Ruby-интерпретатора.

Задания

1. Установите Ruby-интерпретатор на своем компьютере.
2. Напишите Ruby-программу, которая выводит ваши имя и фамилию.

ГЛАВА 2

Быстрый старт

Файлы с исходными кодами этой главы находятся в каталоге *start* сопровождающего книгу электронного архива.

В этой главе мы познакомимся с соглашениями языка Ruby и способами вывода информации в стандартный поток вывода. Изучим ключевые слова, переменные, начнем знакомство с объектно-ориентированными возможностями языка. Кроме того, научимся пользоваться документацией.

2.1. Соглашения Ruby

В Ruby существует большое количество самых разнообразных соглашений. Цель этих соглашений — добиться минимального элегантного кода, в котором отсутствуют повторы.

Программный код «живет» много лет, и все это время он нуждается в обслуживании и модификации. Оформленный по соглашениям код позволяет значительно сократить усилия, время и стоимость поддержки приложения.

Чем меньше объем кода, тем меньше ошибок в нем можно совершить. Использование принципа DRY (Don't repeat yourself, не повторяйся) позволяет исключить ситуации, когда изменения в одной части программы приводят к ошибке в другой.

Давайте продемонстрируем несколько соглашений на примере программы, которая выводит две фразы: `'Hello, world!'` и `'Hello, Ruby!'` при помощи двух вызовов метода `puts` (листинг 2.1).

Листинг 2.1. Некорректный вариант программы. Файл hello_wrong.rb

```
puts('Hello, world!');
puts('Hello, Ruby!');
```

Метод `puts` принимает в качестве аргумента строку и выводит ее в стандартный поток вывода. Аргументы методов перечисляются в круглых скобках, которые следуют за названием. В конце выражения может быть размещена точка с запятой. За-

пустив программу из листинга 2.1 на выполнение, можно убедиться, что она успешно отработает и выведет в консоль две строки.

Однако в Ruby так оформлять программы не принято — все необязательные элементы, которые можно опустить, обычно опускаются. Необязательная точка с запятой всегда опускается. Круглые скобки после названия метода используются только в том случае, если аргументов много или метод сам является аргументом для других методов. Там, где от скобок можно избавиться, от них избавляются (листинг 2.2).

> ***Замечание***
>
> На страницах этой книги мы будем еще не раз останавливаться на соглашениях, принятых в языке. С полным списком соглашений можно ознакомиться по ссылке: **https://github.com/arbox/ruby-style-guide/blob/master/README-ruRU.md**.

Листинг 2.2. Корректный вариант программы. Файл hello_right.rb

```
puts 'Hello, world!'
puts 'Hello, Ruby!'
```

Впрочем, существуют ситуации, когда без точки с запятой обойтись нельзя. Например, когда два выражения располагаются на одной строке (листинг 2.3).

Листинг 2.3. Ситуация, когда точка с запятой обязательна

```
puts 'Hello, world!'; puts 'Hello, Ruby!'
```

Если в программе из листинга 2.3 убрать точку с запятой между двумя вызовами, интерпретатор «запутается» и не сможет разобрать программу.

Впрочем, существует еще одно соглашение: каждое выражение должно быть расположено на отдельной строке. Поэтому точка с запятой в языке Ruby практически никогда не используется. Исключение, пожалуй, составляет только интерактивный Ruby (см. *разд. 3.2*), где удобно создавать длинные однострочные выражения.

2.2. Комментарии

Не все выражения в программе подлежат интерпретации. Иногда в код необходимо добавить разъяснения или в процессе отладки временно исключить из выполнения участок кода. Для создания таких игнорируемых участков в программе предназначены специальные конструкции — *комментарии*.

Самый простой способ создать комментарий — это воспользоваться символом решетки: #. Все, что расположено, начиная с этого символа до конца строки, считается комментарием (листинг 2.4).

Листинг 2.4. Использование комментариев. Файл comment.rb

```
# Вывод строки в стандартный поток
puts 'Hello, world!' # Hello, world!
```

Первая строка в программе полностью игнорируется интерпретатором, т. к. символ решетки расположен на первой позиции строки. Во второй строке комментарий начинается после выражения `puts`.

Интерпретатор читает файл программы сверху вниз, слева направо. Поэтому он сначала выполнит выражение, дойдет до начала комментария и проигнорирует все, что расположено после символа `#`.

В Ruby имеется и многострочный комментарий, который начинается с последовательности `=begin` и завершается `=end` (листинг 2.5).

Листинг 2.5. Использование многострочного комментария. Файл comment_multi.rb

```
=begin
puts 'Hello, world!'
puts 'Hello, Ruby!'
=end
```

Все, что расположено между `=begin` и `=end`, считается комментарием. Современные редакторы позволяют быстро комментировать выделенный участок кода при помощи клавиатурных сокращений. Поэтому в профессиональных программах практически невозможно обнаружить такой тип комментариев. Предпочтение отдается однострочным комментариям.

Существует еще более экзотический способ закомментировать участок программы. Ключевое слово `__END__` обозначает конец Ruby-программы — все, что расположено после него, игнорируется интерпретатором (листинг 2.6).

Листинг 2.6. Использование ключевого слова `__END__`. Файл comment_end.rb

```
puts 'Hello, world!'
puts 'Hello, Ruby!'
__END__
Тут можно располагать все, что угодно
Ruby-интерпретатор считает, что файл закончился на __END__
```

Обратите внимание, что до и после `END` следуют два символа подчеркивания. Строго говоря, `__END__` предназначен не для комментирования, а для создания в программе *секции данных*, извлечь которую можно при помощи предопределенной константы `DATA` (см. *разд. 6.2*).

2.3. Элементы языка

Когда мы приступаем к изучению языка программирования, то знакомимся с элементами языка — строительными кирпичиками, из которых потом будем строить наши программы.

С частью элементов мы уже познакомились. Так, в предыдущих разделах мы затронули: ключевые слова, методы и строковые выражения. Некоторые из этих эле-

ментов уже готовыми предоставляет сам язык программирования, часть придется придумать и разработать самостоятельно.

В языки Ruby различают следующие синтаксические элементы:

- ключевые слова;
- переменные;
- константы;
- объекты;
- классы;
- модули;
- методы;
- операторы.

Некоторые из этих элементов очень тесно связаны друг с другом. На протяжении книги мы подробно рассмотрим их синтаксис и использование на практике. В этой же главе мы лишь кратко охарактеризуем упомянутые элементы.

2.3.1. Ключевые слова

Ключевое слово — это неизменная часть синтаксиса языка, поведение которой задает интерпретатор. Например, выражения можно поместить в составной блок `begin` и `end` (листинг 2.7).

> ***Замечание***
>
> Обратите внимание на отступы вложенных выражений в листинге 2.7. В соответствии с соглашениями в Ruby используются отступы в два пробела.

Листинг 2.7. Ключевые слова `begin` и `end`. Файл keywords.rb

```
begin
  puts 'Hello, world!'
  puts 'Hello, Ruby!'
end
```

Конструкции `begin` и `end` являются ключевыми словами, поведение которых задает интерпретатор Ruby. Если мы попробуем использовать ключевое слово в качестве переменной, интерпретатор не сможет выполнить такую программу (листинг 2.8).

Листинг 2.8. Ошибочное использование ключевого слова. Файл keywords_error.rb

```
begin = 2 + 2
puts begin
```

Запустив программу на выполнение, мы получим сообщение об ошибке:

```
$ ruby keywords_error.rb
keywords_error.rb:1: syntax error, unexpected '='
begin = 2 + 2
keywords_error.rb:2: syntax error, unexpected end-of-input
```

Интерпретатор считает `begin` ключевым словом и не ожидает, что выражение, следующее за ним, будет начинаться со знака равенства.

2.3.2. Переменные

Если мы хотим вычислить выражение 2 + 2 и поместить его в переменную, нам потребуется дать этой переменной какое-либо имя (листинг 2.9).

Листинг 2.9. Создание переменной. Файл variable.rb

```
sum_result = 2 + 2
puts sum_result # 4
```

Результат вычисления выражения 2 + 2 мы помещаем в переменную, дав ей имя `sum_result`. Теперь переменная содержит значение 4, которое мы можем получить в другом выражении, обратившись к переменной ее по имени.

Переменная — это именованная область памяти (рис. 2.1). Обратите внимание, что имя переменной записывается в snake-стиле: строчными (маленькими) буквами с разделением отдельных слов символом подчеркивания.

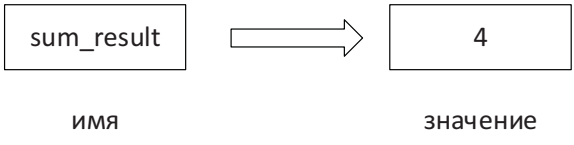


Рис. 2.1. Переменная — это значение в памяти, на которое можно сослаться при помощи имени переменной

В переводе с английского «snake» — «змея». Слова, разделенные символами подчеркивания, напоминают ползущую по земле змею. В противовес snake-стилю существует CamelCase-стиль: `HelloRuby`. В этом стиле слова записываются заглавными буквами. «Camel» в переводе с английского — «верблюд». Название родилось из-за того, что заглавные буквы в нем напоминают горбы верблюда.

В имени переменной могут использоваться буквы, цифры и символ подчеркивания. При этом переменная не может начинаться с цифры, но может начинаться с символа подчеркивания: `var`, `_var`, `first_name`, `fahrenheit451`.

2.3.3. Константы

Константы — это также именованные области памяти, однако, в отличие от переменных, их значение не должно изменяться в ходе выполнения программы. Константы в Ruby всегда начинаются с заглавных букв (листинг 2.10).

Листинг 2.10. Использование константы. Файл constant.rb

```
CONST = 12
puts CONST # 12
```

Помимо собственных констант, можно воспользоваться многочисленными константами, которые предоставляет Ruby. Например, в листинге 2.11 используется

предопределенная константа RUBY_VERSION, которая возвращает текущую версию языка.

Листинг 2.11. Использование константы RUBY_VERSION. Файл ruby_version.rb

```
puts RUBY_VERSION
```

Несмотря на то, что константы не должны менять свое значение, в Ruby это сделать можно (листинг 2.12).

Листинг 2.12. Изменение константы. Файл constant_change.rb

```
CONST = 12
puts CONST # 12
CONST = 14
# constant_change.rb:3: warning: already initialized constant CONST
# constant_change.rb:1: warning: previous definition of CONST was here
puts CONST # 14
```

При попытке изменения значения константы будет выведено предупреждение о том, что значение константы уже определено. Такое нетипичное поведение констант, отличающееся от других языков программирования, связано с тем, что константы в Ruby играют роль *механизма импорта* (см. *главу 6*).

2.3.4. Объекты

Ruby — абсолютно объектно-ориентированный язык, в нем практически все является *объектом*. У объектов есть поведение, которое задается *методами*. Вызывая метод у объекта, мы можем отправлять ему сообщение.

Например, мы можем запросить уникальный идентификатор объекта при помощи метода object_id:

```
'Hello world!'.object_id # 70271671481960
```

Можем проверить, входит ли число в диапазон от 0 до 10 при помощи метода between?:

```
3.between? 0, 10 # true — истина
3.between? 10, 20 # false — ложь
```

Кстати, логические значения true и false, обозначающие во всех современных языках программирования истину и ложь, тоже являются объектами:

```
true.object_id # 120
```

Вызов метода осуществляется через оператор точки (рис. 2.2). Слева от точки всегда располагается объект, или, как еще говорят, *получатель*, справа от точки всегда располагается метод.

Иногда получатель можно не указывать — например, для глобальных методов вроде puts получатель никогда не указывается. Однако в Ruby нельзя вызвать метод

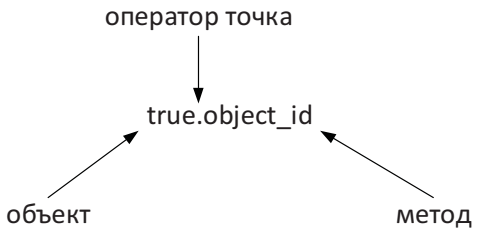


Рис. 2.2. Слева от точки всегда располагается объект, справа — метод

без получателя, пусть даже неявного. При желании мы могли бы указать получатель и при вызове метода `puts`:

```
$stdout.puts 'Hello world!'
```

Разумеется, если можно не указывать необязательный элемент, он никогда не указывается, поэтому более правильно вызывать метод `puts` следующим образом:

```
puts 'Hello world!'
```

2.3.5. Классы и модули

Строковые и числовые объекты создаются при помощи специального облегченного синтаксиса, который поддерживается интерпретатором. Большинство объектов создаются при помощи *класса* — специального объекта, задающего поведение других объектов. Создать объект можно, вызвав метод `new` класса (листинг 2.13).

> ***Замечание***
>
> Обратите внимание, что название классов задаются в CamelCase-стиле, т. е. название класса это еще и константа.

Листинг 2.13. Создание объекта класса `Object`. Файл object.rb

```
obj = Object.new
puts obj.object_id # 70097855352540
```

Для создания объекта мы воспользовались готовым классом `Object`, который предоставляет Ruby. Для этого мы вызывали метод `new`. Таким образом, класс — это обычный объект, в отношении которого мы можем вызывать методы.

В листинге 2.13 мы воспользовались готовым классом, однако при помощи ключевого слова `class` мы можем создавать свои собственные классы (листинг 2.14).

Листинг 2.14. Создание собственного класса `Hello`. Файл class.rb

```
class Hello
end

h = Hello.new
puts h.class # Hello
```

Каждый объект поддерживает метод `class`, при помощи которого можно выяснить класс объекта.

Модули предназначены для организации пространства имен и очень похожи на классы, только без возможностей создания объектов. Более подробно модули и приемы работы с ними мы рассмотрим в *главах 19–21*.

2.3.6. Методы

Методы — это именованные последовательности действий. Метод создается при помощи ключевого слова `def`, за которым следует имя метода и тело метода, которое заканчивается ключевым словом `end`. После имени метода в круглых скобках могут быть перечислены параметры, которые можно опустить, если метод не принимает ни одного параметра (листинг 2.15).

Листинг 2.15. Создание метода `greeting`. Файл method.rb

```
def greeting
  puts 'Hello, world!'
end

greeting
```

Ключевое слов `def` лишь сообщает о том, что метод должен быть определен и зарегистрирован. Исполнение содержимого метода осуществляется лишь тогда, когда интерпретатор встречает в программе название метода. В приведенном примере — это последняя строка программы.

Метод можно добавить и в класс, в результате появляется возможность вызывать методы у объектов этого класса (листинг 2.16).

Листинг 2.16. Размещение метода внутри класса. Файл class_method.rb

```
class Hello
  def greeting
    puts 'Hello, world!'
  end
end

h = Hello.new
h.greeting # Hello, world!
```

Здесь метод `greeting` помещается в теле класса `Hello`, в результате мы можем вызывать метод в отношении объекта `h`.

2.3.7. Операторы

Оператор — это инструкция языка, которая позволяет в краткой и наглядной форме получить результат вычислений. Например, арифметические операторы позволяют записывать математические выражения в форме, привычной со школы:

```
puts 5 + 2 # 7
puts 5 - 2 # 3
puts 5 * 2 # 10
puts 5 / 2 # 2
```

С точки зрения Ruby-интерпретатора, оператор — это обычный метод, для которого Ruby предоставляет специальный синтаксис:

```
puts 5 + 2
puts 5.+ 2
puts 5.+(2)
```

Все три выражения полностью эквивалентны: для объекта `5` вызывается метод `+`, которому в качестве аргумента передается объект `2`. При передаче аргументов методу круглые скобки можно не указывать, кроме того, интерпретатор позволяет не указывать точку перед методами-операторами.

2.4. Вывод в стандартный поток

Каждая программа в современных операционных системах по умолчанию связывается с тремя стандартными потоками: ввода, вывода и ошибок. По умолчанию эти потоки связываются с консолью. Поэтому, когда мы вводим строки при помощи метода `puts`, информация попадает в консоль. Более подробно мы познакомимся с потоками ввода/вывода в *разд. 27.1*.

2.4.1. Вывод при помощи методов *puts* и *p*

Метод `puts` может принимать более одного параметра, которые перечисляются через запятую (листинг 2.17). В этом случае будет уместным использовать круглые скобки.

Листинг 2.17. Передача `puts` нескольких аргументов. Файл puts.rb

```
puts('hello', 'world')
```

Результатом работы программы из листинга 2.17 будут две строки:

```
hello
world
```

Фактически вызов `puts` с несколькими аргументами аналогичен отдельным вызовам метода `puts` с каждым из них.

Особенностью `puts` является добавление перевода строки в конце вывода. Если необходим вывод без перевода строки, можно воспользоваться методом `print` (листинг 2.18).

Листинг 2.18. Использование метода `print`. Файл print.rb

```
print 'Hello, '
puts 'world!'
```

Результатом работы программы из листинга 2.18 будет строка:

```
Hello, world!
```

При выводе объекта в консоль часто бывает трудно определить природу объекта:

```
puts '3' # 3
puts 3 # 3
```

Вывод строки с числом `3` и числа `3` будут полностью эквивалентны. Это может затруднять процесс отладки. Поэтому часто используют метод `inspect` для того, чтобы определить истинную природу объекта:

```
puts '3'.inspect # "3"
puts 3.inspect # 3
```

С использованием метода `inspect` объект выводится как есть — так, как его «видит» Ruby-интерпретатор. Для комбинации `puts` и метода `inspect` существует короткий вариант — метод `p`:

```
p '3' # "3"
p 3 # 3
```

> ***Замечание***
>
> Для более читаемого вывода и для вывода сложных объектов можно использовать метод `pp`. До версии Ruby 2.5 это требовало подключения библиотеки `pretty-printer` при помощи инструкции `require 'pp'`. В настоящее время метод можно использовать непосредственно, как и любой другой метод вывода: `puts`, `print` или `p`.

2.4.2. Экранирование

Иногда в строках, которые заключены в одиночные кавычки, могут встречаться другие одиночные кавычки (листинг 2.19).

Листинг 2.19. Ошибочное формирование строки. Файл escape_wrong.rb

```
str = 'Rubist's world'
puts str
```

Эта программа не сможет выполниться, т. к. интерпретатор не сумеет корректно разобрать строку. В качестве выражения будет выделена последовательность: `str = 'Rubist'`, а остаток фразы: `s world'` приведет к возникновению ошибки:

```
escape_wrong.rb:1: syntax error, unexpected tIDENTIFIER, expecting end-of-input
str = 'Rubist's world'
```

Для решения возникшей проблемы существует несколько вариантов. Один из них заключается в использовании вместо одиночных кавычек — двойных (листинг 2.20).

Листинг 2.20. Использование двойных кавычек. Файл double_quotes.rb

```
str = "Rubist's world"
puts str
```

Двойные и одиночные кавычки — два штатных способа создания строк в Ruby. Часто, когда в строке необходимо использовать одиночные кавычки, строка формируется двойными кавычками, и, наоборот, когда в строке необходимо разместить двойные кавычки — она обрамляется одиночными.

Еще один способ размещения кавычек внутри строки заключается в их *экранировании*. Для этого перед кавычкой размещается символ обратного слеша (листинг 2.21).

Листинг 2.21. Экранирование одиночной кавычки. Файл escape_right.rb

```
str = 'Rubist\'s world'
puts str
```

Экранированию можно подвергать не только кавычки — обратный слеш может не только отменять специальное поведение символов, но и изменять поведение обычных символов на специальное.

В листинге 2.22 мы заменяем метод `puts` на `print`, а в конце строки добавляем последовательность `\n`, которая обозначает перевод строки.

> ***Замечание***
>
> В операционной системе Windows для перевода строки используются два спецсимвола — возврат каретки и перевод строки: `\r\n`.

Листинг 2.22. Экранирование одиночной кавычки. Файл escape.rb

```
print "Rubist's world\n"
```

Наиболее часто используемые последовательности приводятся в табл. 2.1.

Таблица 2.1. *Специальные символы и их значения*

Значение	Описание
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Символ табуляции
`\\`	Обратный слеш
`\"`	Двойная кавычка
`\'`	Одиночная кавычка

2.5. Как пользоваться документацией?

Обязательным навыком любого рубиста является быстрый поиск описания возможностей объектов Ruby. Особенно это важно на начальных этапах освоения языка, когда базовое поведение наиболее часто используемых классов и методов не заучено твердо.

2.5.1. Справочные методы объектов

В отношении объекта можно вызывать не любой метод. В листинге 2.23 приводится пример вызова метода `greeting` у объекта `h` класса `Hello` и объекта `o` класса `Object`.

Листинг 2.23. Ошибка вызова метода `greeting`. Файл method_error.rb

```
class Hello
  def greeting
    puts 'Hello, world!'
  end
end

h = Hello.new
o = Object.new
h.greeting # Hello, world!
o.greeting # method_error.rb:10:in `<main>': undefined method `greeting'
```

Мы успешно можем вызвать метод `greeting` в отношении объекта `h`, однако при вызове такого метода в отношении объекта `o` терпим неудачу, поскольку метод не реализован на уровне класса `Object`.

В таком намеренно упрощенном примере достаточно легко определить, в отношении какого из объектов можно вызывать метод `greeting`. Здесь это очевидно, т. к. класс `Hello` с методом определен непосредственно в файле программы. Однако часто бывает, что класс определен глубоко в системе в какой-либо библиотеке. Например, возможность вызова метода `object_id` у объекта класса `Object` уже не очевидна:

```
o = Object.new
puts o.object_id # 70221864572560
```

Зачастую даже не понятно, к какому классу относится тот или иной объект. Поэтому на практике в первую очередь стараются определить класс, к которому принадлежит объект. Сделать это можно при помощи уже упомянутого в *разд. 2.3.5* метода `class`:

```
2.class # Integer
'hello'.class # String
```

Далее можно исследовать методы, которые допускается вызывать в отношении объекта. Например, для того чтобы выяснить, можно ли вызывать в отношении того или иного объекта метод, можно воспользоваться методом `respond_to?`, передав ему в качестве аргумента строку с названием метода:

```
3.respond_to? 'between?' # true — истина
3.respond_to? 'puts' # false — ложь
```

Более того, объекты предоставляют готовый метод `methods`, который позволяет получить список всех методов, которые можно применить в отношении объекта (листинг 2.24).

Листинг 2.24. Использование метода `methods`. Файл methods.rb

```
puts 3.methods
```

Результатом работы программы из листинга 2.24 будет длинный список методов:

```
%
&
...
instance_exec
__id__
```

Обладая именами класса объекта и метода, всегда можно однозначно определить страницу документации, которая описывает текущее поведение программы.

2.5.2. Консольная справка

При установке Ruby вместе с интерпретатором устанавливается консольная документация, получить доступ к которой можно при помощи утилиты `ri` (ее название является сокращением от ruby index). Если передать утилите имя класса, модуля или метода, будет выведена консольная справка, перемещаться по которой можно при помощи клавиш вверх <↑> и вниз <↓>, для выхода из документации используется клавиша <q>.

> ***Замечание***
>
> Менеджер версий RVM разворачивает утилиту `ri` без документации. В случае RVM для активации документации следует выполнить команду: `rvm docs generate`.

Получить полный список доступных классов можно, передав утилите параметр `--list`:

```
$ ri --list
```

Впрочем, чаще утилите передается название конкретного класса:

```
$ ri Integer
```

Или метода этого класса:

```
$ ri Integer#round
```

В документации Ruby используется особая нотация для обозначения методов объектов и классов:

- `Float#round` — инстанс-методы, которые можно вызывать в отношении объекта: `1.6.round`;
- `Math::sin` — методы классов, которые вызываются без создания объектов: `Math::sin(2)`;
- `Object.object_id` — точкой обозначаются инстанс-методы и методы класса;

> ***Замечание***
>
> Более подробно различия между методами объекта, или, как еще говорят, *инстанс-методами* и *методами класса*, освещаются в *главе 14*.

Помимо консольной утилиты `ri`, можно использовать документацию в графическом интерфейсе — например, Zeal для операционных систем Windows, Linux и Dash в macOS.

2.5.3. Online-документация

В дополнение к документации, получаемой в командной строке, можно воспользоваться online-документацией на сайте **http://ruby-doc.org**. Для этого следует перейти в раздел **Core**, точная ссылка на который зависит от версии Ruby — например, **http://ruby-doc.org/core-2.5.3/**.

На странице документации слева располагается список классов, слева — список методов (рис. 2.3).

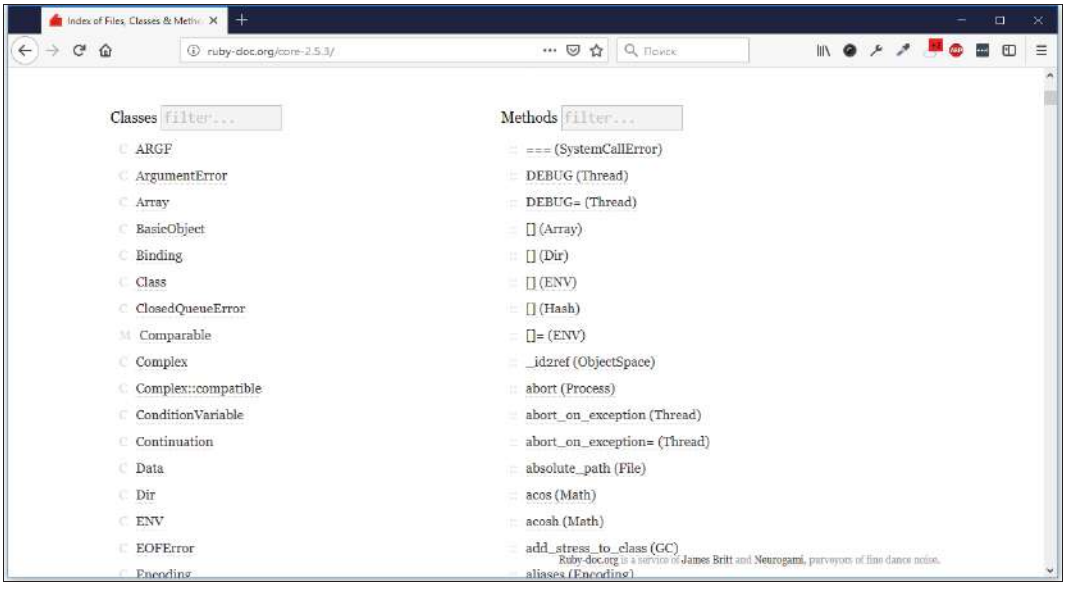


Рис. 2.3. Официальная документация языка Ruby допускает поиск по классам и методам

Если известно название класса, его можно набрать в левом поле поиска, если интересует конкретный метод, его можно поискать в правом поле поиска. На странице класса приводится описание порядка работы с классом, а также подробное описание методов и примеры их использования.

Задания

1. Создайте четыре класса: пользователь, компьютер, сеть, хост. Подберите классам подходящие названия и создайте объекты этих классов.
2. Создайте класс пользователя `User` и снабдите его двумя методами: `fio`, который возвращает имя и фамилию пользователя, и `profession`, который возвращает профессию. Создайте программу, которая демонстрирует работу класса.

ГЛАВА 3

Утилиты и гемы

Файлы с исходными кодами этой главы находятся в каталоге *utils* сопровождающего книгу электронного архива.

Интерпретатор Ruby поставляется с несколькими полезными утилитами — две из них нам уже знакомы: это сам интерпретатор в виде утилиты `ruby` и клиент для доступа к консольной документации `ri`.

Еще больше возможностей предоставляет утилита `gem`, при помощи которой можно загрузить Ruby-библиотеки, гемы и значительно расширить как возможности языка, так и инструментальный набор утилит.

В частности, здесь мы установим гемы `pry`, `rubocop` и `bundler`, которые позволят отлаживать код, следить за его стилистическим оформлением и автоматизировать процесс установки гемов в проект.

Примечание

За исключением первых двух разделов материал этой главы можно при первом чтении пропустить.

3.1. Утилиты

При установке Ruby вместе с интерпретатором языка обязательно устанавливается ряд дополнительных утилит, которые можно обнаружить в папке bin. Список этих утилит представлен в табл. 3.1.

Таблица 3.1. *Утилиты дистрибутива Ruby*

Утилита	Описание
`ruby`	Интерпретатор языка Ruby
`irb`	Интерактивный Ruby
`ri`	Утилита для чтения консольной документации
`rdoc`	Утилита для формирования документации
`rake`	Утилита для выполнения связанных задач на языке Ruby

Таблица 3.1 (окончание)

Утилита	Описание
erb	Шаблонизатор, позволяющий выполнять внутри тестовых файлов Ruby-код, за счет вставки его в тэги <% ... %>
gem	Утилита для установки сторонних библиотек

Как и отмечалось ранее, в предыдущих главах с утилитами ruby и ri мы уже познакомились. В последующих разделах мы рассмотрим работу с другими утилитами.

3.2. Интерактивный Ruby

Для выполнения кода его не обязательно размещать в файле с расширением rb. Интерпретатор Ruby можно запустить в интерактивном режиме при помощи утилиты irb (название утилиты является сокращением от Interactive Ruby):

```
$ irb
 > 2 + 2
=> 4
 > 'Hello world!'
=> "Hello world!"
 >
```

После запуска утилиты изменяется приглашение командной строки, после которого мы можем размещать любой Ruby-код. Нажатие клавиши <Enter> запускает процесс интерпретации. При этом нет необходимости использовать методы вывода: puts, print и p — утилита irb самостоятельно выводит результат вычисления выражения после символа =>.

Впрочем, допускается использование и методов вывода — например, puts:

```
 > puts 'Hello world!'
Hello world!
=> nil
```

В этом случае сначала выводится результат вывода методом puts, а после символа стрелки => — результат вычисления всего выражения: puts 'Hello world!'. В данном случае это объект nil, обозначающий неопределенное значение (подробнее nil освещается в *разд. 4.10*).

При программировании на Ruby следует учитывать, что каждое выражение возвращает какое-либо значение, которое можно использовать далее в программе:

```
hello = (puts 'Hello world')
```

В интерактивном Ruby к объекту применяется метод inspect, а результат выводится после символа стрелки =>:

```
(puts 'Hello world').inspect
```

Метод `inspect` имеется у каждого объекта, за исключением объектов класса `BasicObject`, в чем можно убедиться, попробовав создать объект этого класса в интерактивном Ruby:

```
 > BasicObject.new
(Object doesn't support #inspect)
=>
```

Консоль очень часто используется для экспериментов — например, для проверки какой-либо гипотезы или разработки сложного выражения.

Кроме того, интерактивный Ruby можно использовать для отладки кода. Для этого в коде программы необходимо разместить вызов `binding.irb` (листинг 3.1).

Листинг 3.1. Отладка кода при помощи `irb`. Файл irb.rb

```
class Hello
  def greeting
    puts 'Hello, world!'
  end
end

h = Hello.new
o = Object.new
h.greeting

binding.irb

o.greeting
```

После запуска программы интерпретатор остановит ее выполнение в точке вызова `binding.irb`. Будет выведен код программы в окрестностях точки остановки и предоставлена консоль утилиты `irb`, в которой можно запросить состояние переменных или выполнить какие-либо вычисления:

```
$ ruby irb.rb
Hello, world!

From: irb.rb @ line 11 :

     6:
     7: h = Hello.new
     8: o = Object.new
     9: h.greeting
    10:
 => 11: binding.irb
    12:
    13: o.greeting

>
```

Слева выводятся номера строк в файле программы, которые позволяют сопоставить текущее положение указателя с содержимым файла в текстовом редакторе.

3.3. Шаблонизатор *erb*

Утилита `erb` — простейший шаблонизатор, который позволяет размещать в текстовых файлах Ruby-вставки. Для этого используются два типа тэгов:

- `<% ... %>` — вычисление произвольного Ruby-выражения;
- `<%= ... %>` — подстановка результата вычисления Ruby-выражения вместо тэгов.

Пример использования этих тэгов приводится в листинге 3.2. Как правило, файлам, содержащим Ruby-вставки, присваивается расширение erb.

Листинг 3.2. Программа с erb-вставками. Файл template.erb

```
Выражение 2 + 2 = <%= 2 + 2 %>
Метод puts позволяет вывести результаты вычислений в стандартный поток
<% puts 'Hello, world!' %>
```

Для выполнения Ruby-кода и получения результата достаточно передать файл в качестве параметра утилите `erb`:

```
$ erb template.erb
Hello, world!
Выражение 2 + 2 = 4
Метод puts позволяет вывести результаты вычислений в стандартный поток
```

Утилита анализирует содержимое ERB-файла и последовательно выполняет Ruby-код сверху вниз. Именно поэтому фраза `Hello, world!` выводится первой. Вместо всех вставок `<%= ... %>` подставляются результаты вычислений и в конце выводится содержимое файла с выполненными подстановками.

Шаблоны ERB используются по умолчанию во многих Ruby-фреймворках — например, в Ruby on Rails, — как для формирования представлений, так и для безопасной передачи паролей в конфигурационные файлы.

Впрочем, для представлений чаще задействуются более продвинутые системы шаблонизации, в ряде случаев являющиеся полноценными декларативными языками программирования. Среди наиболее популярных можно отметить Slim и Haml. Установить их в проект можно при помощи утилиты `gem`, которая более подробно рассматривается в *разд. 3.6*.

3.4. Утилита *rake*

Многие задачи выполняются как зависимые друг от друга последовательности действий. Если такие задачи разрабатываются на Ruby, для их выполнения удобно использовать утилиту `rake`, название которой навеяно одноименной утилитой `make` из

мира языка программирования C. Собственно, `rake` и расшифровывается как ruby make.

Для того чтобы воспользоваться утилитой, необходимо создать конфигурационный файл Rakefile, в котором будут размещаться Rake-задачи (листинг 3.3).

Листинг 3.3. Конфигурационный файл `rake`. Файл Rackfile (предварительно)

```
task :load do
  puts 'Hello, world!'
end
```

Пока мы лишь начинаем изучать язык Ruby, поэтому не станем здесь подробно разбирать синтаксические конструкции из листинга 3.3. Сейчас нам достаточно знать, что при помощи метода `task` создается Rake-задача с именем `load`, при выполнении которой выводится фраза: `'Hello, world!'`.

Для того чтобы запустить задачу на выполнение, необходимо передать ее имя утилите `rake`:

```
$ rake load
Hello, world!
```

Если сейчас выполнить утилиту `rake` без параметров — произойдет ошибка:

```
$ rake
rake aborted!
Don't know how to build task 'default' (see --tasks)
```

Дело в том, что утилита ищет по умолчанию задачу `default`. Давайте ее добавим в файл Rackfile (листинг 3.4).

> ***Замечание***
>
> Содержимое *`Rackfile`* — это обычный Ruby-код. Однако из-за того, что в нем исключены круглые и фигурные скобки, Ruby-код выглядит как своеобразный декларативный язык программирования. Подобные мини-языки называют *DSL-языками* (Domain-specific language), или *предметно-ориентированными языками* программирования. Разработка таких языков крайне популярна в Ruby-сообществе.

Листинг 3.4. Rake-задача по умолчанию: `default`. Файл Rackfile (предварительно)

```
task default: :load

task :load do
  puts 'Hello, world!'
end
```

У методов `task` не обязательно должен быть блок с телом. Кроме того, задачи можно связывать в цепочки, чтобы они зависели друг от друга. В листинге 3.4 задача `default` зависит от задачи `load`. Это означает, что при выполнении инструкции `rake default` предварительно будет выполнена задача `load`:

```
$ rake
Hello, world!
```

Задачам можно добавлять описания, для чего используется метод `desc` (листинг 3.5).

Листинг 3.5. Использование метода `desc`. Файл Rackfile (предварительно)

```
desc 'Задача по умолчанию'
task default: :load

desc 'Вывод фразы Hello, world!'
task :load do
  puts 'Hello, world!'
end
```

В том случае, если перед методом `task` размещается DESC-описание, задача появляется в списке задач, которые можно получить при помощи утилиты `rake`, передав ей параметр `-T` или `--task`:

```
$ rake -T
rake default  # Задача по умолчанию
rake load     # Вывод фразы Hello, world
```

Rake-задачи можно переоткрывать: если мы заводим две разные задачи с одним и тем же именем, то при выполнении будут выполнены обе задачи (листинг 3.6).

Листинг 3.6. Переоткрытие Rake-задачи. Файл Rackfile (предварительно)

```
desc 'Задача по умолчанию'
task default: :load

task :load do
  puts 'Hello, world!'
end

task :load do
  puts 'Hello, Ruby!'
end
```

При выполнении задачи `load` из листинга 3.6 отработают обе задачи:

```
$ rake load
Hello, world!
Hello, Ruby!
```

Rake-задачи можно группировать в пространства имен, которые вводятся при помощи метода `namespace` (листинг 3.7). В качестве первого аргумента метод принимает название пространства имен.

Листинг 3.7. Использование пространств имен. Файл Rackfile (окончательно)

```
desc 'Задача по умолчанию'
task default: 'test:load'

namespace :test do
desc 'Вывод фразы Hello, world!'
task :load do
 puts 'Hello, world!'
end
end
```

При запросе списка все Rake-задачи из пространства имен получают дополнительный префикс, совпадающий с именем пространства:

```
$ rake -T
rake default    # Задача по умолчанию
rake test:load  # Вывод фразы Hello, world
```

При вызове задачи так же необходимо указывать этот префикс:

```
$ rake test:load
Hello, world!
```

При помощи пространства имен можно изолировать в том числе и одноименные Rake-задачи. Кроме того, пространство имен позволяет сгруппировать задачи по направлениям: часть задач может обслуживать базу данных, часть — документацию, другие — занимаются генерацией кода.

3.5. Утилита *rdoc*

Изначально утилита `rdoc` предназначена для извлечения документации из исходных кодов Ruby. Документация на сайте **http://ruby-doc.org** формируется именно этой утилитой.

Утилиту `rdoc` можно использовать для извлечения документации из собственного Ruby-кода. Для этого комментарии к нему необходимо оформлять по специальным правилам (листинг 3.8).

> ***Замечание***
>
> Вместо `rdoc` в проектах часто используется более богатый возможностями гем `yard`, установить который можно при помощи утилиты `gem` (см. *разд. 3.6*).

Листинг 3.8. Комментарии для извлечения утилитой `rdoc`. Файл rdoc.rb

```
##
# Класс для вывода приветствия
class Hello
```

```
  ##
  # Выводит в стандартный вывод строку приветствия,
  # интерполируя в нее параметр +row_count+
  #
  # = Пример использования
  #
  #   h = Hello.new
  #   h.geeting('Ruby')
  def greeting(name)
    puts "Hello, #{name}"
  end
end
```

Для того чтобы извлечь документацию из файла rdoc.rb, достаточно передать его утилите `rdoc`:

```
$ rdoc rdoc.rb
```

В ответ будет сформирована папка doc, содержащая всю извлеченную документацию в HTML-формате. Для навигации по документации следует запустить индексный файл index.html в браузере.

3.6. Гемы

Как уже отмечалось ранее, в переводе с английского «ruby» означает «рубин». Соответственно, для обозначения Ruby-библиотек используется термин *гем* (gem), что по-английски — драгоценный камень.

Рассмотренная в *разд. 3.4* утилита `rake` на самом деле является гемом, который поставляется вместе с дистрибутивом Ruby. Исходный код и описание гема можно найти на странице: **https://github.com/ruby/rake**.

Гем оформляется по определенным правилам, упаковывается в пакет и при необходимости регистрируется на сайте **rubygems.org**. В последнем случае гем может быть загружен при помощи утилиты `gem`. Утилита может принимать множество дополнительных команд, самой популярной из которых является `install`, — с ее помощью как раз и можно установить гем.

Например, установить более свежую версию гема `rake` можно, выполнив следующую команду:

```
$ gem install rake
Fetching: rake-12.3.1.gem (100%)
Successfully installed rake-12.3.1
1 gem installed
```

Во время установки утилита посещает сайт **rubygems.org**, ищет указанный гем и загружает его и все его зависимости.

Для поиска гемов можно воспользоваться командой `gem search`:

```
$ gem search rake

*** REMOTE GEMS ***

airake (0.4.5)
...
rake (12.3.1)
...
zotplus-rakehelper (0.0.157)
```

Если дополнительно указать параметр `-l`, поиск гемов будет осуществляться на локальной машине:

```
$ gem search -l rake

*** LOCAL GEMS ***

rake (12.3.1)
```

Таким путем можно убедиться, установлен гем или нет. Поскольку гемы сами зависят друг от друга, они весьма чувствительны к версиям. Поэтому на компьютере может быть установлено несколько версий одного и того же гема. В этом случае в отчете команды `gem search` версии перечисляются через запятую в круглых скобках.

Если необходимо получить полный список гемов, можно воспользоваться командой `gem list`:

```
$ gem list -l

*** LOCAL GEMS ***

bigdecimal (default: 1.3.4)
...
zlib (default: 1.0.0)
```

Для того, чтобы удалить гем, нужно выполнить команду `gem uninstall`:

```
$ gem uninstall rake
```

Работа в командной строке может быть весьма эффективна, но не очень удобна. Чаще всего гемы ищут через веб-интерфейс — в том числе и на сайте **rubygems.org**, на главной странице которого имеется поле поиска.

При выборе гема следует ориентироваться на количество его загрузок, а также на количество звездочек на странице GitHub с исходным кодом, датой последнего обновления и частотой выхода релизов. Если познания Ruby позволяют проанализировать код гема — обязательно следует познакомиться с его организацией, тестовым покрытием и документацией.

Существуют специальные каталоги-подборки с разбивкой гемов по тематикам: базы данных, разработка игр, командная строка и т. д. Наиболее известные среди таких каталогов: **https://awesome-ruby.com/** и **https://www.ruby-toolbox.com/**.

3.6.1. Отладка программ при помощи гема *pry*

При разработке Ruby-программ в коде могут часто возникать ошибки — как синтаксические, так и логические. В *разд. 3.2* мы уже познакомились с отладкой программы при помощи `irb`.

Однако в Ruby-сообществе более популярной является отладка при помощи гема `pry`. Когда вы только начинаете работать с этим гемом, крайне полезно посетить его страницу на GitHub: **https://github.com/pry/pry** и ознакомиться с документацией.

Вместе с гемом `pry` также желательно сразу установить гем `pry-byebug`, который позволяет осуществлять навигацию по коду при помощи команд: `next` и `step`:

```
$ gem install pry
$ gem install pry-byebug
```

Гемы организованы по-разному — часть из них предоставляет утилиты командной строки, некоторые потребуют подключения при помощи метода `require`. Гем `pry` относится к последней категории (листинг 3.9). Возможности `require` более детально обсуждаются в *главе 6*, а в данном случае он просто подключает библиотеку `pry`.

Листинг 3.9. Подключение гема `pry`. Файл pry.rb

```
require 'pry'

class Hello
  def greeting
    puts 'Hello, world!'
  end
end

h = Hello.new
o = Object.new
h.greeting

binding.pry

o.greeting
```

Для создания точки остановки в программу следует добавить вызов `binding.pry`. После запуска программы на выполнение интерпретатор дойдет до точки вызова `binding.pry`, остановит выполнение программы, выведет окрестности точки выполнения и предоставит интерактивный Ruby (см. *разд. 3.2*), в котором можно будет поэкспериментировать с текущими переменными:

```
$ ruby pry.rb
Hello, world!

     8:
     9: h = Hello.new
    10: o = Object.new
    11: h.greeting
    12:
 => 13: binding.pry
    14:
    15: o.greeting

[1] pry(main)>
```

Напомню, что слева выводятся номера строк в файле программы, которые позволяют сопоставить текущее положение указателя с содержимым файла в текстовом редакторе.

Помимо вызова Ruby-выражений, в PRY-консоли можно использовать специальные навигационные команды. Например, чтобы продвинуться в выполнении программы дальше, необходимо ввести команду `next`, если нужно «провалиться» внутрь метода, — команду `step`. Если вывод команд слишком объемен, и не понятно, в какой точке программы мы находимся в текущий момент, можно воспользоваться командой `whereami`, которая выведет код в окрестностях текущей точки останова. Навигационные команды работают лишь в том случае, если был установлен гем `pry-byebug`.

Для того чтобы продолжить выполнение программы в автоматическом режиме, можно воспользоваться либо командой `!!!`, либо клавиатурным сокращением <Ctrl> + <D> (в операционной системе macOS: <Cmd> + <D>).

3.6.2. Контроль стиля кода при помощи гема *rubocop*

В Ruby большую роль играют соглашения — набор правил, которых придерживаются большинство Ruby-разработчиков. Часть из них мы уже рассмотрели. Например, в конце выражения можно ставить точку с запятой, но она никогда не ставится, для отступов в программах всегда используются два пробела и т. д.

Ознакомиться с полным набором соглашений можно по ссылке: **https://github.com/arbox/ruby-style-guide/blob/master/README-ruRU.md**. Соглашений довольно много, и запомнить их все не просто. Более того, необязательно отслеживать нарушения рекомендаций вручную. Предусмотрен специальный гем `rubocop`, который позволяет выполнить эту работу в автоматическом режиме. Домашняя страница гема `rubocop` расположена по ссылке: **https://github.com/rubocop-hq/rubocop**.

> ***Замечание***
>
> Программы вроде `rubocop`, которые анализируют код на предмет поиска ошибок или стилевых несоответствий, называют *линтерами*.

Установить гем rubocop можно утилитой gem, передав ей команду install и название гема:

```
$ gem install rubocop
```

Гем rubocop не требуется подключать к программе, вместо этого он предоставляет одноименную утилиту:

```
$ rubocop --help
```

Для начала проверки достаточно запустить команду rubocop без параметров в папке с Ruby-файлами:

```
$ rubocop
```

В качестве результата утилита выведет список с найденными нарушениями. Часть из них можно устранить автоматически, запустив команду rubocop с параметром -a:

```
$ rubocop -a
```

С некоторыми требованиями rubocop весьма трудно примириться — например, он требует, чтобы все комментарии были оформлены на английском языке. Отключить часть правил rubocop можно через конфигурационный файл .rubocop.yml (листинг 3.10).

Листинг 3.10. Конфигурационный файл гема rubocop. Файл .rubocop.yml

```
AllCops:
  TargetRubyVersion: 2.6
  Exclude:
    - 'tmp/**/*'
Style/AsciiComments:
  Enabled: false
```

Здесь при помощи директивы TargetRubyVersion указывается текущая версия Ruby. В массиве Exclude можно указать, какие папки и вложенные подпапки не должны анализироваться rubocop. Директива Style/AsciiComments позволяет отключить требование оформления комментариев на английском языке.

3.6.3. Управление гемами при помощи *bundler*

Большая часть современного приложения состоит из уже готовых гемов. Для того чтобы не писать каждый раз инструкцию о том, какие гемы необходимо установить, используется менеджер гемов bundler. Этот менеджер сам является гемом, и в его задачу входит управление установкой других гемов. Подробную документацию к гему bundler можно найти на странице: **https://bundler.io/**.

Для установки гема bundler можно воспользоваться утилитой gem install, которой в качестве аргумента передается название гема:

```
$ gem install bundler
```

После установки гема в командной строке становится доступна утилита `bundle`. Для инициализации проекта в его папке необходимо выполнить команду `bundle init`. Гем предоставляет две команды-синонима: `bundler` и `bundle`. Пользоваться можно любой из них:

```
$ bundle init
```

Результатом работы команды становится создание конфигурационного файла Gemfile (листинг 3.11).

> ***Замечание***
>
> Содержимое *`Gemfile`* на самом деле является Ruby-программой. Наряду с *`Rackfile`*, это пример еще одного DSL-языка (Domain-specific language), или предметно-ориентированного языка программирования.

Листинг 3.11. Конфигурационный `bundler`. Файл Gemfile (предварительно)

```
source 'https://rubygems.org'
git_source(:github) {|repo_name| "https://github.com/#{repo_name}" }
```

Первый метод — `source` — позволяет задать источник гема. Как можно видеть, в листинге 3.11 указывается официальный сайт: **rubygems.org**. Некоторые компании разворачивают свои собственные зеркала сайта **rubygems.org** для ускорения загрузки.

Второй метод — `git_source` — позволяет задать псевдоним `github` для загрузки гемов напрямую с GitHub, не прибегая к сайту **rubygems.org**.

Для установки гема в файле Gemfile необходимо разместить метод `gem`, которому в качестве аргумента передается название гема (листинг 3.12).

Листинг 3.12. Регистрация гемов. Файл Gemfile (предварительно)

```
source 'https://rubygems.org'
git_source(:github) {|repo_name| "https://github.com/#{repo_name}" }

gem 'rubocop'
gem 'pry'
```

Для того чтобы выполнить установку гемов, необходимо отдать команду `bundle install` или просто вызвать `bundle` без аргументов:

```
$ bundle install
```

После выполнения команды в папке проекта появляется папка vendor с гемами и дополнительный конфигурационный файл Gemfile.lock. Этот файл автоматически генерируется `bundler` при установке и обновлении гемов, и он не предназначен для ручного редактирования (пример файла Gemfile.lock также находится в каталоге start сопровождающего книгу электронного архива).

Основное назначение файла Gemfile.lock — зафиксировать текущие версии гемов. Обычно папку vendor добавляют в файл .gitignore, чтобы система контроля версий не следила за кодом гемов, которые можно загрузить в любой момент. В то же время файл Gemfile.lock обязательно индексируется — если выполнить команду `bundle install` в папке с готовым файлом Gemfile.lock, будут загружены библиотеки именно тех версий, которые в нем указаны. Такое поведение `bundler` гарантирует одинаковое поведение проекта на компьютерах коллег по команде и при эксплуатации у заказчика.

Если возникает необходимость обновить версию какого-то гема, можно воспользоваться командой `bundle update`, которой передается название гема:

```
$ bundle update pry
```

Версии гемов можно фиксировать на уровне конфигурационного файла Gemfile. Метод `gem` для этого может принимать второй необязательный аргумент, который задает версию гема. Если версия не указывается, используется последняя стабильная версия.

Версии, как правило, состоят из трех цифр: мажорная, минорная и патч-версия (см. рис. 1.3). *Мажорная* версия обычно меняется при кардинальной перестройке гема: `rails` 4.0.0, `rails` 5.0.0 и т. д. *Минорная* версия меняется при релизе новой стабильной версии: 5.1.0. *Патч-версия* меняется при исправлении багов или усовершенствованиях, которые не затрагивают интерфейсы и поведение программного обеспечения: 5.1.1.

Влиять на версии гемов можно в конфигурационном файле Gemfile. Например, последовательность >= требует установки любой версии гема, выше указанной:

```
gem 'web-console', '>= 3.3.0'
```

Можно указать диапазон версий:

```
gem 'listen', '>= 3.0.5', '< 3.2'
```

Или зафиксировать мажорную и минорную версии при помощи последовательности ~>:

```
gem 'rails', '~> 5.1'
```

В приведенном примере допускается версия гема `rails` от 5.1.0 до 5.2.0, не включая последнюю.

Точно так же, как и утилита `gem`, гем `bundler` не только устанавливает сами гемы, но и все его зависимости. Более того, зависимости можно представить при помощи команды `bundle viz`:

```
$ bundle viz
```

> ***Замечание***
>
> Для корректной работы команды `bundle viz` потребуется установить гем `ruby-graphviz` командой:
>
> `gem install ruby-graphviz`.

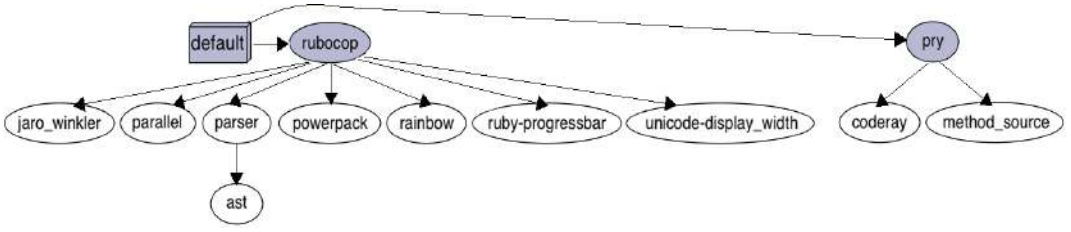


Рис. 3.1. Графическое представление зависимостей гемов

Результатом работы команды будет изображение gem_graph.png с графом зависимостей гемов (рис. 3.1).

Мы установили два гема: `rubocop` и `pry` — они представлены темными эллипсами. Белыми эллипсами показываются зависимости гемов, которые были установлены дополнительно.

Команда `bundle show` позволяет узнать, где расположен гем на уровне файловой системы:

```
$ bundle show rubocop
/Users/i.simdyanov/utils/vendor/bundle/gems/rubocop-0.60.0
```

Команда `bundle open` открывает папку с гемом в редакторе по умолчанию:

```
$ bundle open rubocop
```

Для работы последней команды потребуется прописать запуск редактора из командной строки в переменной окружения: `BUNDLER_EDITOR`.

Получить детальную информацию о геме можно при помощи команды `bundle info`:

```
$ bundle info rubocop
  * rubocop (0.60.0)
       Summary: Automatic Ruby code style checking tool.
       Homepage: https://github.com/rubocop-hq/rubocop
       Path: /Users/i.simdyanov/utils/vendor/bundle/gems/rubocop-0.60.0
```

В отчете команды представлены текущая версия, краткое описание, домашняя страница и расположение гема в локальной файловой системе.

Для подключения гемов в файл проектов можно воспользоваться кодом, представленным в листинге 3.13.

Листинг 3.13. Подключение гемов в проект. Файл bundler.rb

```
require 'rubygems'
require 'bundler/setup'

Bundler.require(:default)
```

Первые два вызова метода `require` подключают `bundler`, последняя строка подключает все гемы, перечисленные в файле Gemfile.

Если в подключении гема нет необходимости, методу `gem` нужно передать дополнительный параметр `require: false`:

```
gem 'rubocop', require: false
```

Этот параметр сообщает о необходимости игнорировать гем при вызове `Bundler.require`.

Многие гемы — такие как `rubocop` или `pry` — необходимы лишь при разработке проекта. При его установке и эксплуатации в них нет надобности. Управлять набором гемов, которые необходимы на разных стадиях жизненного цикла проекта, можно при помощи *окружений*.

Окружения вводятся в файл Gemfile при помощи метода `group`, который принимает в качестве аргумента название окружения (листинг 3.14).

Листинг 3.14. Использование окружений. Файл Gemfile (окончательно)

```
source 'https://rubygems.org'
git_source(:github) {|repo_name| "https://github.com/#{repo_name}" }

group :development do
  gem 'rubocop'
  gem 'pry'
end
```

В листинге 3.14 мы требуем, чтобы гемы `rubocop` и `pry` устанавливались только для development-окружения, которое, как правило, обозначает режим запуска приложения в ходе разработки.

Методу `group` мы можем передать несколько окружений через запятую:

```
group :development, :test do
  ...
end
```

Можно избежать установки гемов из определенной группы — для этого команде `bundle install` передается дополнительный параметр `--without`:

```
$ bundle install --without test
```

Или, наоборот, можно установить гем из определенной группы при помощи параметра `--with`:

```
$ bundle install --with test
```

Если вы принимаете решение разработать свой собственный гем, `bundler` позволяет сгенерировать заготовку для гема при помощи команды `bundle gem`, которой передается название будущего гема:

```
$ bundle gem hello
```

В ответ команда создаст папку hello, содержащую заготовку будущего гема.

Задания

1. Создайте класс пользователя `User`, снабдите его несколькими методами — например: `fio` — для получения фамилии и отчества, `profession` — для получения профессии пользователя. Оформите документацию для класса и методов, после чего извлеките ее при помощи утилиты `rdoc`.
2. По документации познакомьтесь с предопределенным классом `Time` и его возможностями. С использованием этого класса разработайте Rake-задачу, которая выводит текущую дату и время.
3. В интерактивном Ruby (`irb`) умножьте число 245 на 365.

ГЛАВА 4

Предопределенные классы

Файлы с исходными кодами этой главы находятся в каталоге *ruby_classes* сопровождающего книгу электронного архива.

Во многих языках программирования существует понятие *типа*, который задает поведение переменных и обеспечен поддержкой компилятора или интерпретатора языка программирования.

Ruby — полностью объектно-ориентированный язык, в котором нет понятия типа. Вместо него поведение объектов задается *классами*. В этой главе мы познакомимся с предопределенными классами языка Ruby, которые выполняют роль типов, задавая поведение строк, чисел, коллекций.

4.1. Синтаксические конструкторы

В Ruby нет типов — поведение всех объектов задается их классами. У объекта может быть только один класс, узнать который можно при помощи метода `class`:

```
3.class # Integer
```

Позже мы увидим, как поведение объекта и его класса можно менять на лету — один объект может «притвориться» другим.

Свои собственные классы мы можем создавать при помощи ключевого слова `class` (см. *разд. 2.3.5*). Тем не менее существует множество готовых классов, которые Ruby предоставляет программисту. Например, класс `Object`, получить объект которого можно при помощи метода `new`:

```
o = Object.new
puts o.object_id
```

В то же время часть объектов мы можем создавать более простым способом. Например, для создания строки объекта можно воспользоваться одиночными кавычками. Тем не менее, строка — это тоже объект, и он имеет свой собственный класс `String` (листинг 4.1).

Листинг 4.1. Строка — это объект класса `String`. Файл string.rb

```
str = 'Hello, world!'
puts str.class # String
```

Для создания строки мы могли бы воспользоваться методом `new` класса `String`, т. к. это штатный способ создания объектов в Ruby:

```
str = String.new('Hello, world!')
puts str
```

Впрочем, в отношении строк так никто не поступает — такой код избыточен и громоздок. Когда можно выбрать более короткий вариант, всегда отдается предпочтение самому короткому способу достижения цели.

Специальный способ создания объекта при помощи укороченного синтаксиса — например, кавычек в случае класса `String`, называется *синтаксическим конструктором*.

Существуют несколько предопределенных классов языка Ruby, которые предоставляют синтаксические конструкторы (табл. 4.1).

Таблица 4.1. *Синтаксические конструкторы*

Класс	Синтаксический конструктор	Описание
`String`	`'Hello wold!'`	Строки
`Symbol`	`:white`	Символы
`Integer`	`15`	Целые числа
`Float`	`3.14159`	Числа с плавающей точкой
`Range`	`1..10`	Диапазон
`Array`	`[1, 2, 3, 4]`	Массив
`Hash`	`{hello: 'world', lang: 'ruby'}`	Хэш
`Proc`	`->(a, b) { a + b }`	Прок-объекты
`Regexp`	`//`	Регулярное выражение
`NilClass`	`nil`	Неопределенное значение
`TrueClass`	`true`	Логическая истина
`FalseClass`	`false`	Логическая ложь

В последующих разделах мы рассмотрим синтаксические конструкторы и базовые возможности каждого из представленных в табл. 4.1 классов, за исключением классов `Proc` и `Regexp`, возможности которых более подробно освещаются в *главах 12* и *30* соответственно.

4.2. Строки. Класс *String*

Для создания строки предусмотрены два синтаксических конструктора: одиночные и двойные кавычки:

```
"Hello, world!"
'Hello, world!'
```

Несмотря на то, что объекты содержат одно и то же значение, это два разных объекта, в чем можно убедиться, запросив их идентификатор в консоли интерактивного Ruby:

```
 > "Hello, world!".object_id
=> 70107309581720
 > 'Hello, world!'.object_id
=> 70107309477180
```

Самое главное различие двойных и одиночных кавычек заключается в том, что двойные кавычки поддерживают интерполяцию Ruby-выражений при помощи последовательности `#{...}`, а одиночные кавычки — нет:

```
 > '2 + 2 = #{ 2 + 2 }'
=> "2 + 2 = \#{ 2 + 2 }"
 > "2 + 2 = #{ 2 + 2 }"
=> "2 + 2 = 4"
```

Интерполяция начинается с символа решетки, после которого следуют фигурные скобки, в которых можно разместить любое Ruby-выражение. Результат вычисления этого выражения будет вставлен в строку.

Заключать строки в одиночные кавычки стараются, если в строке нет интерполяции. А если интерполяция есть, двойные кавычки используются в любом случае. При последовательном применении этих правил двойные кавычки служат сигналом, что в строке где-то присутствует интерполяция.

4.2.1. Синтаксические конструкторы *%q* и *%Q*

Помимо кавычек, для создания строк можно использовать специальные последовательности: `%q` и `%Q` (листинг 4.2). После этих последовательностей можно указать строку, заключенную в ограничители, не входящие в состав строки. Здесь могут быть использованы как круглые скобки, так и квадратные или фигурные. Впрочем, соглашения требуют именно круглых скобок.

Листинг 4.2. Создание строки при помощи `%q` и `%Q`. Файл string_q.rb

```
puts %q(Hello, world!)
name = 'Ruby'
puts %Q(Hello, #{name}!)
```

Результатом работы программы из листинга 4.2 будут следующие строки:

```
Hello, world!
Hello, Ruby!
```

В отношении интерполяции Ruby-выражений последовательность `%q` является аналогом одиночных кавычек, в то время как `%Q` — двойных. На практике последовательности `%q` и `%Q` используются весьма редко. Вместо них почти всегда ставятся более компактные и наглядные одиночные и двойные кавычки.

4.2.2. *Heredoc*-оператор

Помимо кавычек, для создания строк можно воспользоваться `heredoc`-оператором (листинг 4.3). Оператор начинается с последовательности `<<`, после которой разработчик указывает уникальную последовательность (в примере это `here`). Вместо `here` может использоваться любая другая последовательность. Как только эта последовательность встречается повторно, считается, что строка завершена.

Листинг 4.3. Создание строки при помощи `heredoc`-оператора. Файл heredoc.rb

```
str = <<here
        В строке, которая формируется heredoc-оператором,
        сохраняются переводы строк.
        Это идеальный инструмент для ввода больших текстов.
here
puts str
```

`Heredoc`-оператор позволяет сохранить форматирование — например, отступы и переводы строк. Результатом выполнения программы из листинга 4.3 будут следующие строки:

```
        В строке, которая формируется heredoc-оператором,
        сохраняются переводы строк.
        Это идеальный инструмент для ввода больших текстов.
```

Завершающая последовательность `here` должна располагаться в начале строки. Если ее предварить отступами, они будут включены в последовательность, и Ruby-интерпретатор не сможет корректно разобрать код программы.

Для того чтобы иметь возможность добавлять перед `here` пробельные символы, последовательность `<<` следует заменить на `<<-` (листинг 4.4).

Листинг 4.4. Игнорирование отступов перед `here`. Файл heredoc-.rb

```
str = <<-here
        В строке, которая формируется heredoc-оператором,
        сохраняются переводы строк.
        Это идеальный инструмент для ввода больших текстов.
      here
puts str
```

Результат выполнения программы из листинга 4.4 полностью эквивалентен полученному ранее при выполнении программы из листинга 4.3.

Часто пробелы в начале строк — результат форматирования `heredoc`-оператора. Если имеется необходимость избавиться от начальных отступов в результирующей строке, можно заменить последовательность << на <<~ (листинг 4.5).

Листинг 4.5. Игнорирование отступов в начале `heredoc`-оператора. Файл heredoc~.rb

```
str = <<~here
        В строке, которая формируется heredoc-оператором,
        сохраняются переводы строк.
        Это идеальный инструмент для ввода больших текстов.
      here
puts str
```

Результатом выполнения программы будут следующие строки:

```
В строке, которая формируется heredoc-оператором,
сохраняются переводы строк.
Это идеальный инструмент для ввода больших текстов.
```

По умолчанию `heredoc`-оператор допускает интерполяцию Ruby-выражений (листинг 4.6).

Листинг 4.6. Интерполяция в `heredoc`-операторе. Файл heredoc_interpolate.rb

```
puts <<~here
       2 + 2 = #{ 2 + 2 }
     here
```

Результатом выполнения программы из листинга 4.5 будет строка:

```
2 + 2 = 4
```

Если такое поведение необходимо отменить, то последовательность `here` следует заключить в одиночные кавычки (листинг 4.7).

Листинг 4.7. Отмена интерполяции. Файл heredoc_not_interpolate.rb

```
puts <<~'here'
       2 + 2 = #{ 2 + 2 }
     here
```

Результатом выполнения программы из листинга 4.7 будет строка:

```
2 + 2 = #{ 2 + 2 }
```

4.2.3. Выполнение команд операционной системы

Еще одним типом кавычек, к которым можно прибегать в Ruby, — это обратные кавычки. Заключенные в них строки рассматриваются как команды, которые необ-

ходимо выполнить в командной оболочке. Результат выполнения команды возвращается в виде строки. В листинге 4.8 извлекается список файлов и каталогов текущей операционной системы.

Листинг 4.8. Выполнение команд в обратных кавычках. Файл string_exec.rb

```
`ls -la`
# Windows
# `dir`
```

Здесь используется команда `ls -la`, подходящая для UNIX-подобных операционных систем. В Windows следует использовать аналог этой команды — команду `dir`.

Для обратных кавычек предусмотрена альтернативная форма записи в виде последовательности `%x` (листинг 4.9). После последовательности можно указать строку, заключенную в ограничители, не входящие в состав строки. Здесь могут быть использованы как круглые скобки, так и квадратные или фигурные.

Листинг 4.9. Выполнение команд в %x. Файл string_x.rb

```
puts %x(ls -la)
# Windows
# puts %x(dir)
```

Обратите внимание: в отличие от обратных кавычек, последовательность `%x` не выводит результат выполнения команды в стандартный поток. Поэтому для получения результата выполнения команды необходимо воспользоваться методом `puts`.

4.2.4. Устройство строки

Строки представляют собой последовательности символов, которые нумеруются, начиная с нуля (рис. 4.1). Ранее, до версии Ruby 1.9, под каждый из символов отводился лишь один байт, однако в современном Ruby строки поддерживают многобайтовую кодировку UTF-8. Эта кодировка расширяемая и позволяет использовать в тексте все возможные языки планеты. Для русских символов в UTF-8 отводятся два байта, для английских — один. Впрочем, это не имеет большого значения в современном Ruby — индексы в строках ссылаются на символы, а не на отдельные байты.

К любому символу строки можно обратиться при помощи квадратных скобок. Внутри скобок следует указать индекс символа в строке (листинг 4.10).

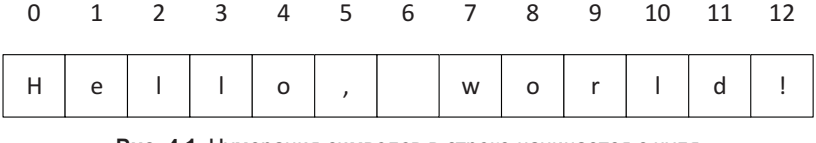


Рис. 4.1. Нумерация символов в строке начинается с нуля

Листинг 4.10. Извлечение символа строки. Файл string_index.rb

```
str = 'Hello, world!'
puts str[0] # H
```

Результатом выполнения программы будет вывод символа `H`. Если мы изменим индекс `0` на `1`, то получим второй символ `e`, и т. д. — вплоть до индекса `12`, который позволит извлечь символ восклицательного знака `!`. Если обратиться к несуществующему символу, например `str[13]`, выражение вернет неопределенное значение `nil`.

4.2.5. Обработка подстрок

В квадратных скобках можно указать два параметра через запятую. В этом случае будет извлекаться подстрока, начиная с индекса, указанного в первом параметре, и длиной, равной второму параметру (листинг 4.11).

Листинг 4.11. Извлечение подстроки. Файл string_substring.rb

```
str = 'Hello, world!'
puts str[7, 5] # world
```

Здесь из строки `'Hello, world!'` извлекается слово `'world'`: символ `w` находится на 7-й позиции, при этом слово `'world'` содержит 5 символов (рис. 4.2).

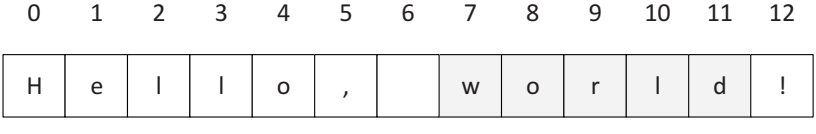


Рис. 4.2. Извлечение подстроки `str[7, 5]`

Заменив запятую на последовательность из двух точек, можно указать диапазон (см. *разд. 4.6*) символов. В листинге 4.12 приводится программа, которая также извлекает подстроку `'hello'`.

Листинг 4.12. Альтернативное извлечение подстроки. Файл string_range.rb

```
str = 'Hello, world!'
puts str[7..11] # world
```

Здесь цифры соответствуют первому и последнему индексам извлекаемой строки. При этом последняя цифра может принимать отрицательные значения. В этом случае отсчет ведется от конца строки. Например, чтобы извлечь все символы, начиная с 7-го до конца строки, мы можем использовать диапазон от `7` до `-1`:

```
str[7..-1] # world!
```

Если потребуется исключить последний символ строки (восклицательного знака), в качестве последней цифры можно использовать -2:

```
str[7..-2] # world
```

Отрицательные символы можно использовать и для первой цифры:

```
str[-6..-2] # world
```

В этом случае мы отсчитываем 6 символов от конца строки для получения левой границы подстроки и 2 символа — для получения ее правой границы (рис. 4.3).

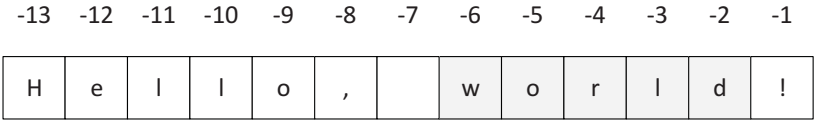


Рис. 4.3. Извлечение подстроки `str[-6, -2]`

Квадратные скобки позволяют не только извлекать подстроки, но и производить в строке замену. В листинге 4.13 в строке `'Hello, world!'` подстрока `'world'` заменяется на `'Ruby'`.

Листинг 4.13. Замена подстроки. Файл string_replace.rb

```
str = 'Hello, world!'
str[7..11] = 'Ruby'
puts str # Hello, Ruby!
```

Используя справа от оператора присваивания пустую строку, не сложно добиться удаления подстрок из строки.

Помимо квадратных скобок, класс `String` предоставляет методы `sub` и `gsub` для замены подстрок:

```
 > 'Hello, world!'.sub('l', '-')
=> "He-lo, world!"
 > 'Hello, world!'.gsub('l', '-')
=> "He--o, wor-d!"
```

Метод `sub` заменяет лишь первую найденную подстроку, в то время как `gsub` осуществляет замену всех вхождений подстроки в строку.

Для того чтобы узнать размер строки, можно воспользоваться методом `size` или альтернативным методом `length`:

```
 > 'Hello, world!'.size
=> 13
 > 'Hello, world!'.length
=> 13
```

Класс `String` предоставляет большое количество разнообразных методов, со многими из которых мы познакомимся на протяжении книги. Однако хотя бы один раз

следует прочитать документацию, посвященную классу `String`, чтобы представлять его возможности.

4.3. Символы. Класс *Symbol*

Символы — уникальные неизменяемые объекты языка Ruby. `Symbol` — один из немногих классов, объекты которого не получится создать при помощи метода `new`, он просто не определен на уровне класса, и попытка его использования завершается ошибкой:

```
Symbol.new
NoMethodError (undefined method `new' for Symbol:Class)
```

Создать символ можно только при помощи синтаксического конструктора, разместив перед идентификатором символ двоеточия (листинг 4.14).

Листинг 4.14. Создание символа. Файл symbol.rb

```
p :white
```

Тогда как переменная представляет собой пару «имя-значение», то символ — это просто «имя» (рис. 4.4).

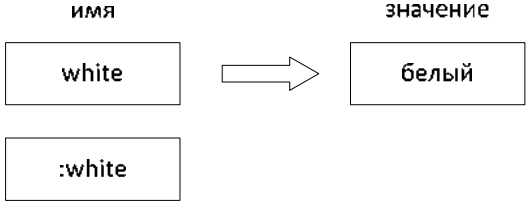


Рис. 4.4. Переменная — это пара «имя-значение», символ — лишь одно «имя»

Если в программе встает задача определить белый цвет, это можно сделать несколькими способами. Например, создав локальную переменную `white`:

```
white = 'белый'
```

или определив константу:

```
WHITE = 1
```

Однако, если цель просто ввести понятие белого цвета — самым коротким способом будет воспользоваться символом:

```
:white
```

> ***Замечание***
>
> В символьных языках, например в LISP или Prolog, существуют похожие синтаксические конструкции, которые называют *атомами*. Понятие *атом* очень точно характеризует природу таких объектов, поскольку символ Ruby представляет лишь одно состояние, представленное его именем. Фактически это и есть настоящие «константы» языка Ruby.

Символы — это неизменяемые объекты. Будучи один раз заведены, они остаются неизменными на всем протяжении работы программы. Они гораздо быстрее обрабатываются, чем строковые объекты, поэтому их применяют во всех случаях, где не требуется изменение строки.

Убедиться в том, что символы не меняются, можно, запросив идентификатор объекта при помощи метода `object_id`:

```
 > :white.object_id
=> 1295068
 > :white.object_id
=> 1295068
```

В отличие от строк, создав символ один раз, интерпретатор в дальнейшем использует его без создания нового объекта:

```
 > 'white'.object_id
=> 70287240535060
 > 'white'.object_id
=> 70287249159480
```

Символы можно преобразовать в объект строки при помощи метода `to_s`, и наоборот — строку можно преобразовать в символ при помощи метода `to_sym`:

```
 > :white.to_s
=> "white"
 > 'white'.to_sym
=> :white
```

Наиболее типичным использованием символов является обозначение методов — например, в методе `respond_to?`, позволяющем выяснить, можно ли применять метод в отношении объектов:

```
 > 'white'.respond_to? :to_sym
=> true
```

Полный список методов объекта также возвращается в виде списка символов:

```
 > 'white'.methods
=> [:include?, ..., :instance_eval, :instance_exec, :__id__, :__send__]
```

Символы могут содержать не только буквы английского алфавита, в них можно использовать практически любые элементы, включая пробелы и знаки препинания. В этом случае содержимое символа заключается в кавычки:

```
:"Hello, world!"
:'()()'
:'123_456_789'
```

4.4. Целые числа. Класс *Integer*

Целые числа представлены классом `Integer`. Они так же, как и символы, являются неизменяемыми, и их нельзя создать при помощи метода `new`. Единственный способ создания объектов этого класса — воспользоваться синтаксическими конструкция-

ми, самая простейшая из которых представляет собой привычное целое число — например, 42.

> ***Замечание***
>
> До версии Ruby 2.4 для целых чисел использовались два класса: `Fixnum` — для коротких и `Bignum` — для больших целых чисел. В Ruby 2.4 эти два класса были объединены в один и переименованы в `Integer`.

Числа могут быть положительными и отрицательными. Для ввода отрицательного числа, как и в арифметике, мы предваряем его символом «минус»: `-`. Перед положительными числами допускается символ `+`, который на практике всегда опускается (листинг 4.15).

Листинг 4.15. Использование знака в числах. Файл integer_sign.rb

```
puts -42 # -42
puts +42 # 42
puts 42  # 42
```

Для простоты восприятия тысячные диапазоны в числах разделяются символом подчеркивания (листинг 4.16).

Листинг 4.16. Разделение тысячных диапазонов в числах. Файл integer_format.rb

```
puts 2000000 + 1900200     # 3900200
puts 2_000_000 + 1_900_200 # 3900200
```

В подавляющем большинстве случаев в программах приходится иметь дело с привычными десятичными числами. Однако в силу того, что компьютеры используют двоичную логику, для работы с компьютерным представлением числа чаще удобнее задействовать не десятичную систему счисления, а кратную двум: двоичную, восьмеричную, шестнадцатеричную.

Если в десятичной системе счисления у нас 10 цифр, в двоичной их только две: 0 и 1. Для задания числа в двоичной системе счисления оно предваряется префиксом `0b`:

```
puts 0b1010101 # 85
```

Использование в таком числе цифр, отличных от 0 и 1, приводит к возникновению ошибки. Компьютер «видит» числа именно в бинарном виде. Как будет показано в *разд. 7.6*, такое представление числа очень полезно при работе с поразрядными операторами.

Числа в восьмеричной системе исчисления могут содержать цифры в диапазоне от 0 до 7, для их создания используется префикс `0o`:

```
puts 0o755 # 493
```

Восьмеричные числа интенсивно используются, например, для задания прав доступа в UNIX-подобных операционных системах (см. *главу 28*).

В шестнадцатеричных числах допускаются цифры от 0 до 9, а также символы `a`, `b`, `c`, `d`, `e`, `f`, которые представляют недостающие цифры: 10, 11, 12, 13, 14 и 15 (рис. 4.5).

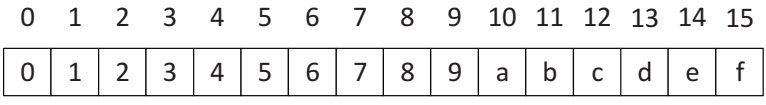


Рис. 4.5. Шестнадцатеричная система счисления

Для создания числа в шестнадцатеричной системе счисления оно предваряется префиксом `0x`:

```
puts 0xffcc00 # 16763904
```

Шестнадцатеричные числа интенсивно используются для задания адресов памяти, цветовых представлений и везде, где требуется более компактная по сравнению с десятичной форма записи целого числа.

Помимо синтаксических конструкторов, получить двоичные, восьмеричные и шестнадцатеричные числа можно из десятичного числа при помощи метода `to_s`. Обычно метод `to_s` используется без аргументов и предназначен для преобразования объектов в строку. В случае чисел допускается необязательный параметр, который задает систему счисления, в которую нужно преобразовать число (листинг 4.17).

Листинг 4.17. Преобразование системы счисления. Файл integer_convert.rb

```
puts 242.to_s(2)  # 11110010
puts 242.to_s(8)  # 362
puts 242.to_s(16) # f2
```

4.5. Вещественные числа. Класс *Float*

За вещественные числа, или числа с плавающей точкой, отвечает класс `Float`. В этом классе так же не получится воспользоваться методом `new`. Для создания вещественного числа придется задействовать один из синтаксических конструкторов.

Числа с плавающей точкой имеют две формы записи. Обычная форма совпадает с принятой в арифметике — например, 346.1256. *Экспоненциальная* форма позволяет представить числа в виде произведения мантиссы 3.461256 и соответствующей степени числа 10. Для цифр меньше нуля степень числа 10 является отрицательной. Так, число 0.00012 в экспоненциальной форме может быть записано как 1.2×10^{-4} (рис. 4.6).

В компьютерных программах нет возможности использовать символы верхнего регистра, поэтому конструкцию $\times10$ записывают в виде символа *e*, после которого указывается значение степени (листинг 4.18).

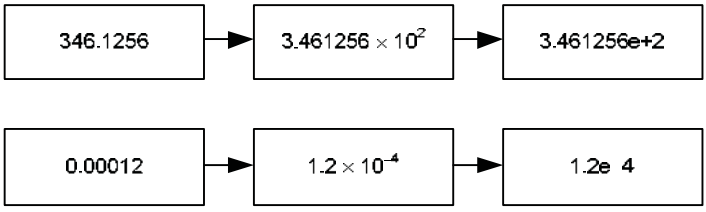


Рис. 4.6. Представление числа с плавающей точкой

Листинг 4.18. Экспоненциальная форма записи числа. Файл float_exponent.rb

```
puts 0.00012     # 0.00012
puts 1.2e-4      # 0.00012
puts 346.1256    # 346.1256
puts 3.461256e+2 # 346.1256
```

Под число с плавающей точкой отводится 8 байтов, что обеспечивает диапазон значений от $\pm 2.23\times10^{-308}$ до $\pm 1.79\times10^{308}$.

При выходе за допустимый диапазон вместо числа выводится константа `Infinity`, символизирующая бесконечность. Любые операции с таким числом опять возвращают `Infinity` (листинг 4.19).

Листинг 4.19. Бесконечные числа и операции с ними. Файл float_infinity.rb

```
puts 1.8e307           # 1.8E+307
puts 1.8e308           # Infinity
puts 1.8e308 - 1.0e307 # Infinity
```

Помимо бесконечности, класс `Float` поддерживает недопустимые числа, которые обозначаются при помощи константы `NaN` (листинг 4.20). Любые операции с таким числом опять возвращает `NaN`.

Листинг 4.20. Бесконечные числа и операции с ними. Файл float_nan.rb

```
puts 0 / 0.0     # NaN
puts 1 - 0 / 0.0 # NaN
```

Для проверки факта, является ли число бесконечным или недопустимым, класс `Float` предоставляет два метода: `infinite?` и `nan?` (листинг 4.21).

Листинг 4.21. Проверка чисел. Файл float_check.rb

```
number = 42.0
infpos = 100 / 0.0
infneg = -100 / 0.0
nan = 0 / 0.0
```

```
p number.infinite? # nil
p infpos.infinite? # 1
p infneg.infinite? # -1

p number.nan? # false
p inf.nan? # true
```

Метод `infinite?` возвращает `nil`, если перед нами конечное число, `1` — если мы имеем дело с положительной бесконечностью `Infinity` и `-1` — если объект является отрицательной бесконечностью `-Infinity`.

Метод `nan?` возвращает `false` в случае обычного числа, и `true` — если число недопустимо.

Этим двум методам противоставляется метод `finit?`, который проверяет, является ли число конечным и допустимым.

> ***Замечание***
>
> Стандартная библиотека Ruby предоставляет классы для работы с рациональными, комплексными числами и матрицами. Для этого используются классы `Rational`, `Complex` и `Matrix`. Их рассмотрение выходит за рамки книги.

В арифметических операциях с участием целых и вещественных чисел результат неявно преобразуется в вещественное число (листинг 4.22).

Листинг 4.22. Неявное преобразование к вещественному числу. Файл float_cast.rb

```
puts 2.class         # Integer
puts 2.0.class       # Float
puts (2 + 2.0).class # Float
```

Неявное преобразование особенно важно в выражениях с участием деления, т. к. операция деления / с участием целых и вещественных чисел ведет себя по-разному. При делении одного целого числа на другое результатом так же является целое число, поэтому дробная часть отбрасывается:

```
puts 7 / 2 # 3
```

В случае вещественных чисел, результат так же является вещественным числом:

```
puts 7.0 / 2.0 # 3.5
```

Если хотя бы один из аргументов является вещественным числом — результат так же является вещественным:

```
puts 7 / 2.0 # 3.5
```

Однако, используя метод `to_i`, можно явно преобразовать вещественное число в целое, а метод `to_f` позволяет совершить обратное преобразование целого в вещественное (листинг 4.23).

Листинг 4.23. Явное преобразование чисел. Файл float_explicit_cast.rb

```
puts 2.0.to_i # 2
puts 2.to_f   # 2.0
```

4.6. Диапазоны. Класс *Range*

Класс `Range` предназначен для организации диапазонов. Этот класс имеет два синтаксических конструктора:

- `1..5` — диапазон, включающий правую и левые границы;
- `1...5` — диапазон, включающий правую, но исключающий левую границу.

На рис. 4.7 представлена развернутая схема диапазонов, а в листинге 4.24 приводится пример создания диапазона.

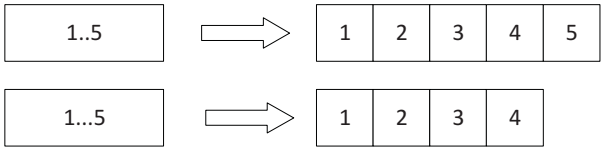


Рис. 4.7. Синтаксические конструкторы диапазонов `Range`

Листинг 4.24. Создание диапазона. Файл range.rb

```
p 1..5         # 1..5
p (1..5).class # Range
```

Диапазоны можно создавать и при помощи конструктора класса `Range`:

```
p Range.new(1, 5)       # 1..5
p Range.new(1, 5, true) # 1...5
```

Впрочем, на практике предпочтение почти всегда отдается более коротким синтаксическим конструкторам.

Объекты диапазона поддерживают множество самых разнообразных методов. Например, мы можем извлечь первый элемент диапазона при помощи метода `first`, а последний — при помощи метода `last` (листинг 4.25).

> ***Замечание***
>
> Методы `first` и `last` имеют синонимы `begin` и `end`.

Листинг 4.25. Получение границ диапазона. Файл range_edges.rb

```
puts (1..5).first # 1
puts (1..5).last # 5
```

При помощи метода `include?` можно проверить, входит ли объект в диапазон (листинг 4.26).

Листинг 4.26. Проверка вхождения в диапазон. Файл range_include.rb

```
range = 1..5
p range.include? 3 # true - истина
p range.include? 7 # false - ложь
```

Начиная с версии Ruby 2.6, доступны бесконечные диапазоны, в которых не указывается правая граница (листинг 4.27).

Листинг 4.27. Бесконечные диапазоны. Файл range_endless.rb

```
p (1..).include? 100500 # true
p (1..).include? -1     # false
```

Отсутствие ограничения диапазона допускается только справа, левая граница в диапазонах обязательна.

Метод `cover?` схож по назначению с методом `include?` — в случае простых объектов методы ведут себя очень похоже (листинг 4.28).

Листинг 4.28. Использование метода `cover?`. Файл range_cover.rb

```
range = 1..5
p range.cover?(3)      # true
p range.cover?(10)     # false

p range.include?(2..3) # false
p range.cover?(2..3)   # true
p range.cover?(3..7)   # false
```

Как видно из листинга 4.28, диапазоны можно сравнивать друг с другом (рис. 4.8). Такое поведение доступно, начиная с Ruby 2.6.0.

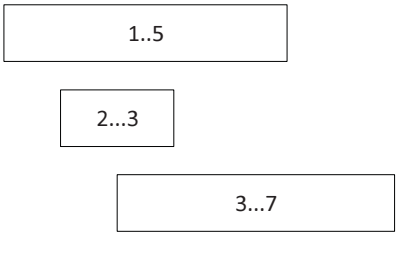


Рис. 4.8. Сравнение диапазонов друг с другом

4.7. Массивы. Класс *Array*

Массивы — это индексированные коллекции. Они очень напоминают строки, только вместо символов у них коллекция произвольных объектов (рис. 4.9).

Более подробно массивы рассматриваются в *главе 23*.

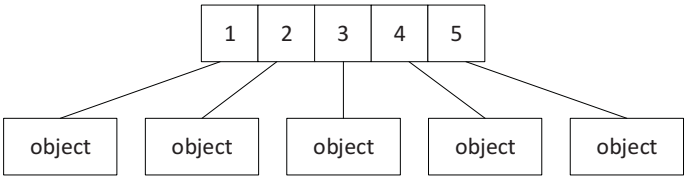


Рис. 4.9. Массив — индексированная коллекция объектов

4.7.1. Создание массива

Самый простой способ создания массива — воспользоваться синтаксическим конструктором «квадратные скобки»: `[]`. Внутри квадратных скобок размещаются элементы через запятую (листинг 4.29).

Листинг 4.29. Создание массива. Файл array.rb

```
p [1, 2, 3, 4, 5]
```

Альтернативным способом создания массива является вызов метода `new` класса `Array`.

```
 > Array.new
=> []
```

На практике предпочитают более компактные квадратные скобки. Однако, как будет показано в *разд. 14.2.2*, есть несколько случаев, в которых использование метода `Array::new` для создания массива является единственным возможным вариантом.

4.7.2. Операции с массивами

Все операции с квадратными скобками, которые использовались в отношении строк, справедливы и в отношении массивов (листинг 2.30).

Листинг 4.30. Создание массива. Файл array_slice.rb

```
arr = [1, 2, 3, 4, 5]
p arr[0]    # 1
p arr[-1]   # 5
p arr[2, 2] # [3, 4]
p arr[2..3] # [3, 4]
```

Индексация в массивах также начинается с нуля, кроме того, допускается обратный отсчет от правой границы массива при помощи отрицательных индексов.

Элементами массива могут выступать любые объекты языка Ruby: числа, строки или даже другие массивы:

```
[1, 'hello', 3, ['first', 'second']]
```

4.7.3. Синтаксические конструкторы *%w* и *%i*

Отдельно следует остановиться на массивах, которые состоят только из строк:

```
colors = [
  'красный', 'оранжевый', 'желтый', 'зеленый',
  'голубой', 'синий', 'фиолетовый'
]
```

Создать такой массив можно более коротким способом, используя синтаксический конструктор `%w` (листинг 4.31).

Листинг 4.31. Создание массива при помощи `%w`. Файл array_w.rb

```
colors = %w[красный оранжевый желтый зеленый голубой синий фиолетовый]
p colors
```

Конструктору `%w` можно передать строку, заключенную в ограничители, в качестве которых по соглашениям выступают квадратные скобки. Внутри ограничителей строка разбивается по пробельным символам, включающим, помимо пробела, символы табуляции и перевода строки:

```
 > colors = %w[красный оранжевый
               желтый  зеленый
               голубой синий
               фиолетовый]
=>["красный", "оранжевый", "желтый", "зеленый", "голубой", "синий",
"фиолетовый"]
```

В случае, если элементы должны содержать пробелы, можно прибегнуть к экранированию пробела при помощи символа обратного слеша:

```
 > %w(Hello,\ world! Hello,\ Ruby!)
=> ["Hello, world!", "Hello, Ruby!"]
```

Как было отмечено в *разд. 4.2.1*, у конструктора `%q` имеется вариант с прописной буквой `%Q`, в котором допускается интерполяция Ruby-выражений. Аналогичная ситуация и в случае конструктора `%w`, для которого предусмотрен вариант `%W` с возможностью интерполяции.

> ***Замечание***
>
> В отличие от последовательностей `%q` и `%Q`, которые задействуются для создания строк весьма редко, все последовательности для создания массивов интенсивно используются на практике.

В качестве элементов массива часто вместо строк используются символы — особенно, в случае перечислений. Например, цвета радуги можно представить массивом символов:

```
colors = [:red, :orange, :yellow, :green, :blue, :indigo, :violet]
```

Для создания массива символов предусмотрен специальный синтаксический конструктор `%i` (листинг 4.32).

Листинг 4.32. Создание массива при помощи `%i`. Файл array_i.rb

```
colors = %i[red orange yellow green blue indigo violet]
p colors
```

В отношении этого конструктора действуют те же правила, что и в отношении конструктора `%w`. Все пробельные символы рассматриваются как разделители, допускается экранирование пробелов при помощи обратного слеша, а для интерполяции Ruby-выражений предусмотрен вариант со строчной буквой `%i`.

4.8. Хэши. Класс *Hash*

Хэш предназначен для хранения коллекций пар «ключ-значение». В качестве ключа и значения могут выступать любые объекты Ruby (рис. 4.10).

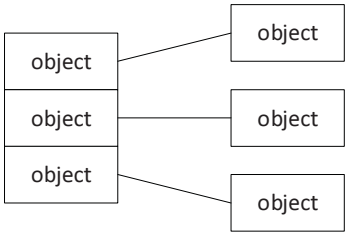


Рис. 4.10. Хэш — коллекция пар «ключ-значение»

Для того чтобы создать хэш, можно воспользоваться синтаксическим конструктором «фигурные скобки»: `{}` (листинг 4.33).

Листинг 4.33. Создание хэша. Файл hash.rb

```
h = { 'first' => 'hello', 'second' => 'world' }
puts h['second'] # world
```

Обращаться к элементам хэша можно по ключам. Так как в качестве ключей могут выступать любые объекты Ruby, очень часто на их месте используются символы:

```
h = { :first => 'hello', :second => 'world' }
```

В таком виде хэши с участием символов оформлялись до версии Ruby 1.9. В современном Ruby для оформления таких хэшей используется новый синтаксис (листинг 4.34).

Листинг 4.34. Использование символов в хэшах. Файл hash_symbols_keys.rb

```
h = { first: 'hello', second: 'world' }
puts h[:first]
```

Синтаксис, представленный в листинге 4.34, является более предпочтительным (подробно хэши рассматриваются в *главе 24*).

4.9. Логические объекты *true* и *false*

Любой современный язык программирования поддерживает работу с логическими значениями, которые могут принимать только два состояния: «истина» и «ложь». Это связано с тем, что наши компьютеры построены на электрических схемах, где очень просто реализовать обработку состояний присутствия или отсутствия заряда, потенциала. Поддержка большего количества состояний значительно усложняет устройство компьютера.

Для обозначения «истины» служит объект `true`, который присутствует в единственном экземпляре:

```
 > true
=> true
 > true.class
=> TrueClass
```

Объект класса `TrueClass` нельзя создать при помощи метода `new`. Интерпретатор Ruby гарантирует, что объект `true` класса `TrueClass` всегда присутствует в единственном экземпляре.

Для обозначения ложного состояния применяется объект `false`, для которого также гарантируется уникальность:

```
 > false
=> false
 > false.class
=> FalseClass
```

Как правило, объекты `true` и `false` интенсивно используются для сравнения, логических вычислений и в конструкциях ветвления:

```
 > 3 == 2 + 1
=> true
 > 4 == 5
=> false
```

Более подробно логические операторы и конструкции ветвления будут рассмотрены в *главе 8*.

4.10. Объект *nil*

Объект `nil` предназначен для обозначения неопределенного, отсутствующего значения. Точно так же, как логические объекты `true` и `false`, объект `nil` присутствует в Ruby-программах в единственном экземпляре:

```
 > nil
=> nil
 > nil.class
=> NilClass
```

Объект `nil` получается в том случае, если мы обращаемся к несуществующей переменной или элементу массива, хэша (листинг 4.35).

> ***Замечание***
>
> До версии Ruby 2.4 для обозначения неопределенного, истинного и ложного состояний допускалось использование соответственных синонимов: `NIL`, `TRUE` и `FALSE` в верхнем регистре. В настоящий момент использование верхнего регистра при обращении к объектам признано устаревшим и не рекомендуется.

Листинг 4.35. Обращение к несуществующему элементу массива. Файл nil.rb

```
arr = [1, 2, 3, 4, 5]
p arr[0]  # 1
p arr[10] # nil
```

Многие операции с участием `nil`, такие как сложение, вычитание, умножение, деление, приводят к ошибкам. При интерполяции в строку `nil` рассматривается как пустая строка.

Объект `nil`, наряду с `false`, в логических выражениях рассматривается как «ложь». Все остальные объекты и выражения Ruby рассматриваются как «истина».

Задания

1. Пусть переменная `pi` содержит значение 3.1415926535. Напишите программу, которая выводит значение числа с точностью до второго знака после точки (3.14).
2. Составьте массив, состоящий из названий дней недели на русском языке. Выведите каждый день недели на отдельной строке.
3. Создайте хэш `colors`, который в качестве ключей содержит символы с названиями цветов радуги на английском языке, а в качестве значений — соответствующие им русские названия цветов.
4. В сутках 24 часа, составьте диапазоны утренних, дневных, вечерних и ночных часов.
5. Создайте массив строк, содержащий цвета радуги. Напишите программу, которая выводит случайный элемент массива.

6. Пусть в двумерной системе координат есть две точки: `{x: 3, y: 7}` и `{x: -1, y: 5}`. Вычислите расстояние между двумя точками по формуле квадрата расстояния.
7. По документации Ruby изучите методы класса `String`. Подберите метод, который переворачивает строку. Например, превращает строку `'Hello'` в `'olleH'`.
8. Пусть имеется строка с начальными и конечными пробелами: `'  hello world  '`. Напишите программу, которая убирает пробелы из начала и конца строки (`'hello world'`).
9. Пусть имеются две строки: `'hello'` и `'world'`. Получите из них одну строку: `'hello world'`.

ГЛАВА 5

Переменные

Файлы с исходными кодами этой главы находятся в каталоге *variables* сопровождающего книгу электронного архива.

Переменные — это элементы языка, имеющие имя и значение. В качестве значения всегда выступает Ruby-объект, обратиться к которому можно по имени переменной.

В Ruby различают несколько типов переменных, которые отличаются синтаксисом и областью видимости.

Существуют предопределенные переменные, которые язык Ruby предоставляет готовыми. Однако подавляющее большинство переменных разработчик вводит в программу самостоятельно, стараясь подобрать им «говорящие» имена, которые будут способствовать пониманию логики программы.

5.1. Типы переменных

В Ruby всего четыре типа переменных, краткая характеристика которых представлена в табл. 5.1.

Таблица 5.1. *Типы переменных*

Переменная	Описание
`local`	Локальная переменная
`$global`	Глобальная переменная
`@object`	Переменная объекта, или инстанс-переменная
`@@klass`	Переменная класса

5.1.1. Локальные переменные

До этого момента мы использовали в основном *локальные переменные* — самые короткоживущие переменные, на пути которых стоит множество барьеров. Одним

из основных барьеров является граница метода, которую локальная переменная не может пересечь никак, — только через параметры или возвращаемые значения.

В листинге 5.1 приводится пример программы, в которой переменной с именем `x` на разных этапах выполнения присваиваются разные значения.

> ***Замечание***
> Подробно создание собственных методов рассматривается в *главе 9*.

Листинг 5.1. Ограничение области видимости локальной переменной. Файл local.rb

```
def say_bye
  x = 'переменная x из say_bye'
end

def start
  x = 'переменная x из start'
  puts x  # переменная x из start
  say_bye
  puts x  # переменная x из start
end

x = 'переменная из глобальной области видимости'
puts x # переменная из глобальной области видимости
start
puts x # переменная из глобальной области видимости
```

Когда Ruby-интерпретатор встречает ключевое слово `def`, он лишь определяет метод, не выполняя код из тела метода. Затем в глобальной области приведенной программы вызывается метод `start`, который в свою очередь вызывает метод `say_bye`. То есть в глобальной области и в каждом из методов определена своя переменная `x` с уникальным значением. Несмотря на то, что по мере выполнения программы мы пытаемся изменить значение переменной, она остается неизменной на каждом из уровней: внутри метода или глобальной области. Таким образом, мы имеем дело с тремя разными переменными, которые называются одинаково — `x`.

Локальные переменные не могут пересекать границу метода (рис. 5.1). Если локальная переменная создается на уровне метода, время ее жизни определяется участком от момента ее определения до выхода из метода. Если такая переменная будет определена за границами метода — это будет совершенно другая переменная.

Для получения списка локальных переменных предназначен метод `local_variables`, который возвращает массив символов с именами всех локальных переменных (листинг 5.2).

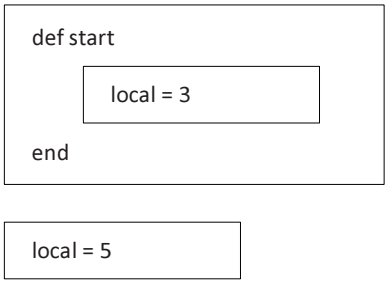


Рис. 5.1. Локальная переменная

Листинг 5.2. Получение списка локальных переменных. Файл local_variables.rb

```
greeting = 'Hello, world!'
number = 100_000

p local_variables # [:greeting, :number]
```

5.1.2. Глобальные переменные

Глобальные переменные начинаются с символа `$` и обладают наибольшей властью, т. к. для них не существует границ, в отличие от остальных типов переменных. К такому типу переменных можно обратиться в любой точке программы.

Если вместо локальной переменной `x` в программе из листинга 5.1 использовать глобальную переменную `$x`, поведение программы изменится (листинг 5.3).

Листинг 5.3. Глобальная переменная не ограничена. Файл global.rb

```
def say_bye
  $x = 'переменная $x из say_bye'
end

def start
  $x = 'переменная $x из start'
  puts $x # переменная $x из start
  say_bye
  puts $x # переменная $x из say_bye
end

$x = 'переменная из глобальной области видимости'
puts $x # переменная из глобальной области видимости
start
puts $x # переменная $x из say_bye
```

Переменная `$x` в листинге 5.3 в любой точке программы — это одна и та же переменная. Для глобальных переменных не существует границ, и обратиться к ней

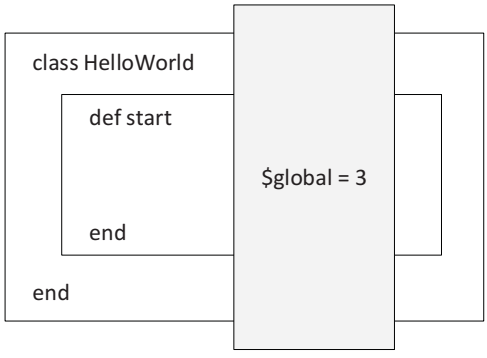


Рис. 5.2. Глобальная переменная

можно из любой точки программы (рис. 5.2). Переменные такого типа существуют все время, пока выполняется программа.

Из-за того, что у глобальных переменных такая широкая область видимости, их стараются не использовать, поскольку отлаживать программы с участием этих переменных очень трудно.

Чем объемнее проект, тем больше точек для установки значений глобальной переменной. В методы и архитектуру проекта могут вкрасться ошибки, и в случае глобальной переменной будет весьма сложно локализовать точку программы, которая приводит к ошибке. Кроме того, становится трудно воспроизвести условия возникновения ошибки. Поэтому в профессиональной разработке — не только на языке Ruby — создания глобальных переменных стараются избегать.

Тем не менее в Ruby-программах можно встретить предопределенные глобальные переменные, которые вводятся интерпретатором. Получить полный список собственных и предопределенных глобальных переменных можно при помощи метода `global_variables` (листинг 5.4).

Листинг 5.4. Список глобальных переменных. Файл global_variables.rb

```
$x = 'переменная из глобальной области видимости'
p global_variables
```

Результат выполнения программы из листинга 5.4 может выглядеть следующим образом:

```
$ ruby global_variables.rb
[:$-I, :$LOAD_PATH, :$", :$LOADED_FEATURES, :$PROGRAM_NAME, :$-W, :$DEBUG,
:$-d, :$0, :$@, :$!, :$stdin, :$stdout, :$stderr, :$>, :$<, :$., :$x,
:$FILENAME, :$*, :$-l, :$-i, :$-a, :$-p, :$SAFE, :$_, :$~, :$?, :$$, :$;, :$-F,
:$&, :$`, :$', :$+, :$=, :$KCODE, :$-K, :$,, :$/, :$-0, :$\, :$VERBOSE, :$-v,
:$-w, :$:]
```

Все переменные в этом списке, кроме `$x`, являются предопределенными. В следующих разделах будет рассмотрена лишь часть предопределенных переменных. С их полным описанием можно познакомиться в официальной документации языка Ruby по ссылке: **http://ruby-doc.org/core-2.5.3/doc/globals_rdoc.html**.

5.1.2.1. Предопределенная переменная *$LOAD_PATH*

Одна из таких переменных — это `$LOAD_PATH`, которая содержит список путей для поиска библиотек и классов Ruby. В листинге 5.5 приводится пример программы, которая выводит содержимое переменной.

> ***Замечание***
>
> Для `$LOAD_PATH` предусмотрен короткий синоним `$:`.

Листинг 5.5. Список путей к библиотекам `$LOAD_PATH`. Файл load_path.rb

```
puts $LOAD_PATH
```

Переменная `$LOAD_PATH` содержит массив, передача которого методу `puts` приводит к выводу каждого элемента массива на отдельной строке.

```
$ ruby load_path.rb
/Users/i.simdyanov/.rvm/gems/ruby-2.5.3@global/gems/did_you_mean-1.2.0/lib
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/site_ruby/2.5.0
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/site_ruby/2.5.0/
                                                        x86_64-darwin15
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/site_ruby
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/vendor_ruby/2.5.0
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/vendor_ruby/2.5.0/
                                                        x86_64-darwin15
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/vendor_ruby
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/2.5.0
/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/lib/ruby/2.5.0/x86_64-darwin15
```

Многие библиотеки и фреймворки расширяют список путей для поиска библиотек. Например, так поступает фреймворк Ruby on Rails. В своих программах тоже можно модифицировать `$LOAD_PATH`, например, добавив в список свой собственный путь (листинг 5.6).

Листинг 5.6. Расширение `$LOAD_PATH`. Файл load_path_extends.rb

```
$LOAD_PATH << '.'
puts $LOAD_PATH
```

Так как в переменной `$LOAD_PATH` хранится массив, мы воспользовались оператором `<<`, чтобы добавить в него новый элемент (более подробно операторы и методы массивов будут рассмотрены в *главе 23*).

5.1.2.2. Предопределенная переменная *$stdout*

Глобальная переменная `$stdout` ссылается на текущий поток вывода, связанный по умолчанию с консолью. Именно в этот поток отправляется информация, которая выводится методом `puts`.

Имея доступ к переменной $stdout, мы можем перехватить поток вывода и направить его, например, в файл. В листинге 5.7 при помощи класса StringIO создается новый поток, который присваивается глобальной переменной $stdout.

Листинг 5.7. Перенаправление стандартного потока вывода в файл. Файл stdout.rb

```
$stdout = StringIO.new
puts 'Hello, world!'
File.write('output.log', $stdout.string)
```

После изменения $stdout весь вывод в программе направляется в новый поток вывода. Поэтому фраза 'Hello, world!' в консоли уже не выводится. Далее при помощи предопределенного класса File открывается новый файл output.log, куда записывается все, что было накоплено во время вывода в программе.

5.1.2.3. Предопределенная переменная *$PROGRAM_NAME*

Глобальная переменная $PROGRAM_NAME позволяет получить название процесса (листинг 5.8).

Листинг 5.8. Получение имени текущего потока. Файл program_name.rb

```
puts $PROGRAM_NAME
```

Запуск программы на выполнение вернет название текущего потока:

```
$ ruby program_name_get.rb
program_name_get.rb
```

Полученные результаты не очень впечатляют, т. к. программа в листинге 5.8 весьма незамысловата и не включает множество Ruby-файлов. Гораздо любопытнее, что при помощи глобальной переменной $PROGRAM_NAME можно изменять имя текущего потока.

Пусть в командной оболочке запущен интерактивный Ruby (irb):

```
$ ps | grep irb
54623 ttys000    0:00.21 irb
55045 ttys001    0:00.00 grep irb
```

Процесс irb имеет значение pid, равный 54623. В UNIX-подобной операционной системе можно узнать параметры процесса, отфильтровав вывод утилиты ps по pid при помощи параметра -p:

```
$ ps -p 54623
  PID TTY           TIME CMD
54623 ttys000    0:00.21 irb
```

Если теперь в интерактивной оболочке поменять название процесса на 'Hello, world':

```
 > $PROGRAM_NAME = 'Hello, world!'
=> "Hello, world!"
```

название процесса в операционной системе также будет изменено:

```
$ ps -p 54623
  PID TTY           TIME CMD
54623 ttys000    0:00.22 Hello, world!
```

5.1.3. Инстанс-переменные

Глобальные переменные имеют слишком широкую область видимости. Поэтому на практике вместо них чаще используют *переменную объекта*, или *инстанс-переменную*. Название таких переменных начинается с одного символа `@`.

Эти переменные являются «глобальными» в рамках одного объекта. Можно записать значение в переменную в одном методе объекта и прочитать в другом методе того же объекта.

Основное назначение инстанс-переменных — сохранять состояние объекта. У каждого объекта свой собственный набор таких переменных (рис. 5.3).

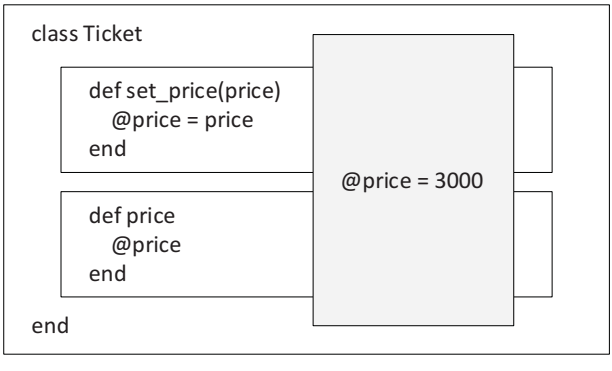


Рис. 5.3. Инстанс-переменная

Для того чтобы оценить возможности инстанс-переменных, создадим класс билета `Ticket`, который будет предоставлять своим объектам два метода: дату проведения мероприятия `date` и цену `price` (листинг 5.9).

Листинг 5.9. Класс `Ticket`. Файл ticket.rb

```
class Ticket
  def date
    '01.10.2019'
  end

  def price
    2000
  end
end

first = Ticket.new
second = Ticket.new
```

```
puts 'Первый билет'
puts "Цена: #{first.date}"
puts "Дата: #{first.price}"

puts 'Второй билет'
puts "Цена: #{second.date}"
puts "Дата: #{second.price}"
```

Сейчас программа не очень удобна для дальнейшего сопровождения. Цена и дата проведения мероприятия размещены непосредственно в классе `Ticket`. Поэтому все объекты этого класса возвращают одинаковые даты и цены. Для того чтобы снабдить каждый из билетов своими собственными датами и ценами, необходимо «научить» объекты сохранять состояние.

В решении этой задачи могут помочь инстанс-переменные. Добавим в класс `Ticket` метод `set_price`, который принимает единственный параметр `price` и сохраняет его значение в инстанс-переменную `@price`:

```
class Ticket
  def date
    '01.10.2019'
  end

  def set_price(price)
    @price = price
  end

  def price
    @price
  end
end
```

После инициализации инстанс-переменной `@price` в методе `set_price` мы можем использовать ее в других методах класса `Price`. Например, вернуть значение инстанс-переменной в методе `price`.

Для сохранения даты мероприятия можно по аналогии ввести инстанс-переменную `@date`. Теперь билетам можно назначить разные цены (листинг 5.10).

> ***Замечание***
>
> Четыре представленных в классе `Ticket` метода можно сократить до одной строки (см. *разд. 14.3.2*).

Листинг 5.10. Использование инстанс-переменных. Файл instance.rb

```
class Ticket
  def set_date(date)
    @date = date
  end
```

```
  def date
    @date
  end

  def set_price(price)
    @price = price
  end

  def price
    @price
  end
end

first = Ticket.new
second = Ticket.new

first.set_price(2000)
first.set_date('31.10.2019')

second.set_price(3000)
second.set_date('02.12.2019')

puts 'Первый билет'
puts "Цена: #{first.date}"
puts "Дата: #{first.price}"

puts 'Второй билет'
puts "Цена: #{second.date}"
puts "Дата: #{second.price}"
```

Результатом работы программы из листинга 5.10 будут следующие строки:

```
Первый билет
Цена: 01.10.2019
Дата: 2000
Второй билет
Цена: 01.10.2019
Дата: 3000
```

Для получения списка инстанс-переменных можно воспользоваться методом `instance_variables` (листинг 5.11). В качестве результата метод возвращает массив символов.

Листинг 5.11. Получение списка инстанс-переменных. Файл instance_variables.rb

```
...
first = Ticket.new

first.set_price(2000)
first.set_date('31.10.2019')

p first.instance_variables # [:@price, :@date]
```

5.1.4. Переменные класса

Переменные, которые начинаются с двух символов `@@`, называются *переменными класса*. Это мини-глобальные переменные в рамках объектов одного класса. Такая переменная является общей для всех объектов одного класса — изменение в одном объекте приводит к изменению переменной для других объектов (рис. 5.4).

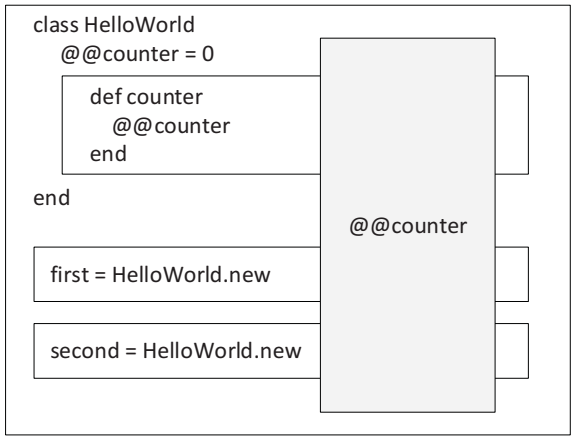


Рис. 5.4. Переменная класса

Замечание

В С-подобных языках для обозначения переменных класса используется термин *статическая переменная*.

В листинге 5.12 определяется класс `HelloWorld`, который содержит переменную класса `@@counter`, инициализированную нулевым значением. В класс добавлены метод `counter_set` — для установки значения `@@counter` и метод `counter` — для считывания переменной класса.

Листинг 5.12. Использование переменной класса. Файл klass.rb

```
class HelloWorld
  @@counter = 0

  def counter
    @@counter
  end

  def set_counter(counter)
    @@counter = counter
  end
end

first = HelloWorld.new
first.set_counter 10
```

```
second = HelloWorld.new
puts second.counter # 10
```

Значение счетчика устанавливается в объекте `first`, а извлечение с последующим выводом в стандартный поток осуществляется уже из объекта `second`.

Для извлечения списка переменных классов предназначен метод `class_variables`. Как и остальные методы для получения списка переменных, метод возвращает массив символов (листинг 5.13).

Листинг 5.13. Извлечение переменных класса. Файл class_variables.rb

```
...
p HelloWorld.class_variables # [:@@counter]
```

5.2. Присваивание

Переменная инициируется при помощи оператора присваивания `=`. Слева от этого оператора располагается переменная, справа — какой-либо объект Ruby или выражение, возвращающее объект. Проще всего выполнить операцию присваивания в консоли интерактивного Ruby:

```
 > number = 1
=> 1
```

Тут переменной с именем `number` присваивается объект `1`. Выражение присваивания возвращает в качестве результата присвоенное значение, поэтому его можно использовать справа от операции присваивания:

```
 > num = (number = 1)
=> 1
```

Здесь значение `1` получит не только переменная `number`, но и переменная `num`. Более того, круглые скобки можно опустить:

```
 > num = number = 1
=> 1
```

При создании объекта в интерактивном Ruby или при использовании метода `p` выводится строковое представление объекта:

```
 > first = Object.new
=> #<Object:0x00007fa465214d80>
```

Представление состоит из двух частей: названия класса — в данном случае `Object`, и адреса в оперативной памяти: `0x00007fa465214d80`, по которому расположен объект. Адреса при каждом новом запуске программы будут различаться.

Таким образом, переменная — это элемент программы, а объект — это структура, расположенная в оперативной памяти компьютера (рис. 5.5). Переменная выступает ссылкой на объект. В программе мы манипулируем легковесной переменной —

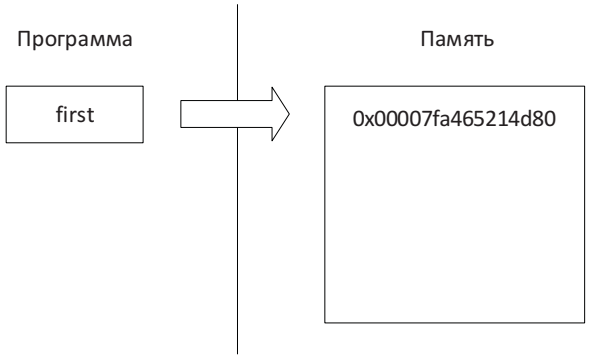


Рис. 5.5. Переменная в программе и объект в оперативной памяти

можем передавать ее внутрь методов, присваивать другим переменным, не перемещая сам объект из одной области памяти в другую.

Если создать новый объект, он будет расположен в новом участке оперативной памяти:

```
 > first = Object.new
=> #<Object:0x00007fa465214d80>
 > second = Object.new
=> #<Object:0x00007fa4651fc1e0>
```

Если переменная перестает ссылаться на объект, утилизацией памяти объекта занимается *сборщик мусора*. Этот процесс полностью скрыт от разработчика, и ему не требуется заботиться о выделении и возвращении памяти операционной системе.

Ссылочная природа переменных имеет значение, когда мы присваиваем или сравниваем переменные друг с другом. Сравнить переменные можно при помощи оператора сравнения, который записывается двумя символами «равно»:

```
 > second == first
=> false
```

Как видим, сравнение возвращает `false` (ложь) — интерпретатор считает переменные разными. Две переменные: `first` и `second` — при сравнении рассматриваются как две разные переменные, т. к. они ссылаются на два разных участках оперативной памяти (рис. 5.6).

Если переменные будут ссылаться на один и тот же объект, то эти переменные будут рассматриваться как равные друг другу (рис. 5.7).

Если присвоить переменной `first` значение переменной `var`, то сравнение этих двух переменных будет возвращать значение `true` (истину):

```
 > var = first
=> #<Object:0x00007fa465214d80>
 > var == first
=> true
```

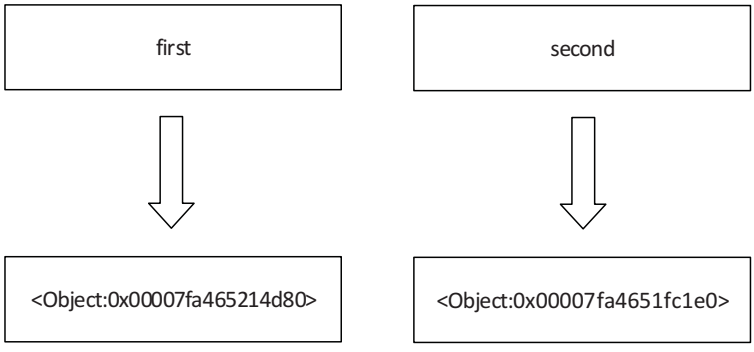


Рис. 5.6. Переменные не равны, если ссылаются на разные объекты

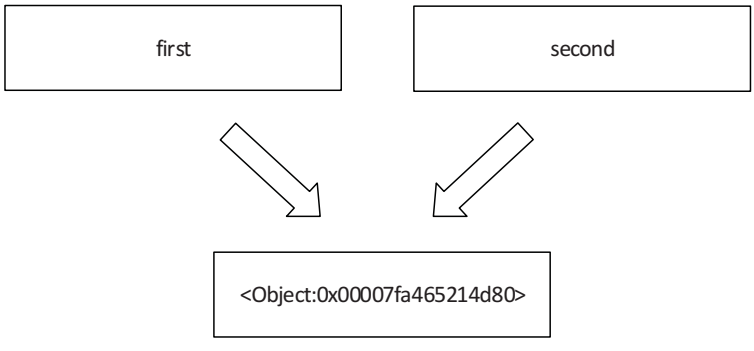


Рис. 5.7. Переменные равны, если ссылаются на один и тот же объект

Кроме того, сравнив адреса в оперативной памяти, на которые ссылаются переменные, мы можем убедиться, что это две разные ссылки на один и тот же объект:

```
 > first
=> #<Object:0x00007fa465214d80>
 > var
=> #<Object:0x00007fa465214d80>
```

5.3. Клонирование

Иногда при присваивании переменных требуется избежать ситуации, когда они ссылаются на один и тот же объект. В этом случае прибегают к операции *клонирования*.

> ***Замечание***
>
> Помимо метода `dup`, язык Ruby предоставляет метод `clone`. Отличие в поведении `dup` и `clone` заключается в особенностях работы с «замороженными» объектами (см. *главу 22*).

Клонирование заключается в создании копии объекта перед тем, как его присвоить переменной. Для этого можно использовать метод `dup`:

```
 > first = Object.new
=> #<Object:0x00007fa465214d80>
 > var = first.dup
=> #<Object:0x00007fa4651caf00>
```

Здесь видно, что адреса объектов различаются, и если теперь попробовать сравнить их, можно убедиться, что это разные объекты:

```
 > var == first
=> false
```

Важно отметить, что метод `dup` осуществляет *поверхностное копирование* объекта. То есть, если в составе объекта имеются ссылки на другой объект, эти ссылки не подвергаются клонированию.

Для демонстрации поверхностного копирования создадим класс радуги `Rainbow`, содержащий инстанс-переменную `@color`, которая содержит массив с цветами радуги (листинг 5.14).

Листинг 5.14. Поверхностное копирование объекта класса `Rainbow`. Файл rainbow.rb

```
class Rainbow
  def set_colors(colors)
    @colors = colors
  end
  def colors
    @colors
  end
end

rainbow = Rainbow.new

colors = %i[red orange yellow green blue indigo violet]
rainbow.set_colors(colors)

copy = rainbow.dup

p copy.colors

p rainbow.object_id # 70315732997580
p copy.object_id    # 70315732997540

p rainbow.colors.object_id # 70315732997560
p copy.colors.object_id    # 70315732997560
```

Объекту класса `Rainbow` доступно два метода: при помощи `set_colors` можно установить массив цветов, а при помощи метода `colors` — прочитать его. И в листинге 5.14 создается объект `rainbow`, в который устанавливается массив цветов, и тут же при помощи метода `dup` создается клонированный объект `copy`.

Каждый объект Ruby снабжен методом `object_id`, который возвращает идентификатор, назначенный объекту интерпретатором. При сравнении объектов чаще удобнее использовать именно этот идентификатор, а не адрес в оперативной памяти. Из листинга 5.14 видно, что объекты `rainbow` и `copy` обладают разными идентификаторами: метод `dup` действительно создал копию объекта `rainbow`, а массив `colors` внутри этих объектов ссылается на один объект, который не был клонирован (рис. 5.8).

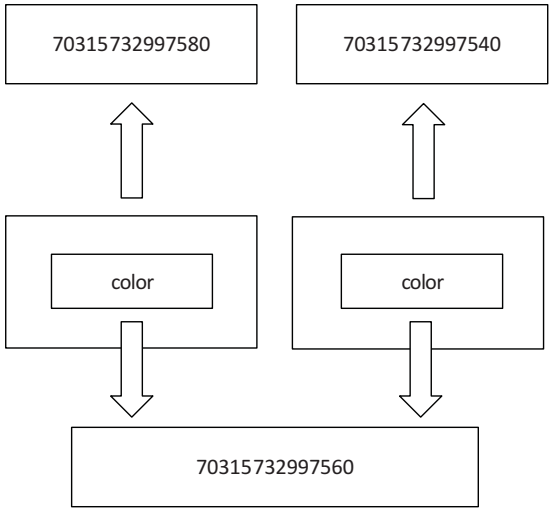


Рис. 5.8. Поверхностное клонирование объекта

Задания

1. Создайте класс пользователя `User`, объекты которого сохраняют фамилию, имя и отчество пользователя. Напишите программу, которая создает объект `user` класса `User`. Поместите в объект ваши фамилию, имя и отчество. Используя объект `user`, выведите их в консоль.
2. Создайте класс точки в двумерной системе координат `Point`. Создайте две точки с координатами (3, 6) и (-1, 5). Вычислите расстояние между точками.
3. Напишите программу, которая подсчитывает количество элементов в массиве `$LOAD_PATH`.
4. Создайте класс автомобиля `Car`. Какие инстанс-переменные и методы следует в него добавить? Создайте несколько объектов легковых и грузовых автомобилей.

ГЛАВА 6

Константы

Файлы с исходными кодами этой главы находятся в каталоге *constants* сопровождающего книгу электронного архива.

Константы в языке Ruby сильно отличаются поведением от аналогичных конструкций в других языках программирования. Основное назначение констант — импортировать объекты. Именно константы извлекаются методами `require` и `require_relative` из файлов и библиотек.

Помимо констант, которые определяет пользователь, Ruby предоставляет большое количество предопределенных констант, интенсивно используемых на практике.

6.1. Создание и определение констант

Как мы уже упоминали в *разд. 2.3.3*, все элементы языка, которые начинаются с заглавной буквы, называется *константами*. Мы можем создавать как свои собственные константы, так и использовать предопределенные константы, которые предоставляют язык Ruby и сторонние гемы. В листинге 6.1 путем обращения к предопределенной константе `RUBY_VERSION` извлекается текущая версия Ruby, а также создается константа с именем `CONST`.

Листинг 6.1. Использование констант. Файл constants.rb

```
CONST = 12
puts CONST        # 12
puts RUBY_VERSION # 2.6.0
```

По соглашениям константы простых базовых объектов — таких как числа, строки, массивы, хэши (см. *главу 4*) — должны полностью состоять из заглавных букв.

Константы в CamelCase-стиле допустимы, но они «зарезервированы» за классами и модулями: `HelloWorld`, `Object`, `Kernel`, `BasicObject`.

В отличие от других языков программирования, значения констант в Ruby можно изменять:

```
 > CONST = 1
=> 1
 > CONST = 2
(irb):15: warning: already initialized constant CONST
(irb):14: warning: previous definition of CONST was here
=> 2
 > CONST
=> 2
```

В случае изменения константы интерпретатор выдаст предупреждение о том, что такая константа уже существует, но, тем не менее, значение будет изменено.

В отличие от переменных, которые можно определять в любой точке Ruby-программы, константы нельзя определять внутри методов (листинг 6.2). Метод может вызываться многократно, что будет приводить к переопределению значения константы. Ruby-интерпретатор предотвращает такую ситуацию.

Листинг 6.2. Константы нельзя определять внутри методов. Файл const_define.rb

```
def hello
  COUNT = 1
end

hello
```

Попытка вызова программы из листинга 6.2 завершается сообщением об ошибке:

```
const_define.rb:2: dynamic constant assignment
```

6.2. Предопределенные константы

В табл. 6.1 представлен ряд готовых констант, которые предоставляет интерпретатор Ruby.

Таблица 6.1. *Предопределенные константы Ruby*

Константа	Описание
RUBY_VERSION	Версия Ruby
RUBY_RELEASE_DATE	Строка с датой релиза Ruby
RUBY_PLATFORM	Строка с идентификатором операционной системы
ARGV	Аргументы, переданные программе
ENV	Переменные окружения
STDOUT	Поток стандартного вывода
STDIN	Поток стандартного ввода
STDERR	Поток ошибок
DATA	Содержимое Ruby-программы после ключевого слова __END__

Каждой программе могут быть переданы параметры, которые можно получить при помощи константы ARGV (листинг 6.3).

Листинг 6.3. Использование константы ARGV. Файл argv.rb

```
p ARGV
```

Передача программе аргументов в командной строке приводит к тому, что они попадают в массив ARGV:

```
$ ruby argv.rb do something --help
["do", "something", "--help"]
```

Предопределенная константа ENV содержит хэш с переменными окружения (листинг 6.4).

Листинг 6.4. Использование константы ENV. Файл env.rb

```
p ENV
```

По умолчанию выводится список всех доступных переменных окружения, которых может быть весьма много. Если нам необходима какая-то одна переменная, для извлечения ее значения можно воспользоваться квадратными скобками. В листинге 6.5 извлекается переменная окружения PATH, которая присутствует во всех современных операционных системах. В этой переменной окружения размещаются пути к каталогам, в которых осуществляется поиск исполняемых файлов.

Листинг 6.5. Извлечение переменной окружения PATH. Файл path.rb

```
p ENV['PATH']
```

В разных операционных системах разделители каталогов в переменной окружения PATH могут различаться. Так в UNIX-подобных операционных системах в качестве разделителя выступает двоеточие :, в Windows — точка с запятой ;.

```
$ ruby path.rb
"/Users/i.simdyanov/.rvm/gems/ruby-2.5.3/bin:/Users/i.simdyanov/.rvm/gems/
ruby-2.5.3@global/bin:/Users/i.simdyanov/.rvm/rubies/ruby-2.5.3/bin:/
Users/i.simdyanov/.rvm/bin:/usr/local/Cellar/pyenv-virtualenv/1.1.0/
shims:/Users/i.simdyanov/.pyenv/shims:/Users/i.simdyanov/.nvm/versions/node/
v7.10.1/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Users/
i.simdyanov/Downloads:/Users/i.simdyanov/golang/bin:/usr/local/
opt/go/libexec/bin"
```

Забегая вперед, давайте разобьем содержимое полученной строки на отдельные пути. Для этого мы применим к строке ENV['PATH'] метод split, передав ему в качестве аргумента двоеточие ':' (листинг 6.6).

Листинг 6.6. Преобразование PATH в массив. Файл path_list.rb

```
puts ENV['PATH'].split(':')
```

Результатом выполнения программы из листинга 6.5 будет список путей из переменной окружения PATH:

```
$ ruby path_list.rb
/Users/i.simdyanov/.rvm/gems/ruby-2.5.3/bin
/Users/i.simdyanov/.rvm/gems/ruby-2.5.3@global/bin
...
/Users/i.simdyanov/Downloads
/Users/i.simdyanov/golang/bin
/usr/local/opt/go/libexec/bin
```

Каждая консольная программа снабжается по умолчанию тремя открытыми потоками ввода/вывода: ввода, вывода и ошибок. Еще одна константа, часто использующаяся на практике, — STDOUT, которая ссылается на поток глобального вывода.

> ***Замечание***
>
> Константа STDOUT очень похожа на предопределенную глобальную переменную $stdout (см. *разд. 5.1.2.2*). Тем не менее, это отдельная константа. Помимо константы STDOUT, Ruby предоставляет константу для стандартного потока ввода STDIN и константу для стандартного потока вывода STDERR (см. *главу 27*).

В *главе 5* при помощи глобальной переменной $stdout перехватывался и направлялся в файл стандартный поток. Для того чтобы восстановить вывод в консоль после работы с файлом, можно воспользоваться константой STDOUT (листинг 6.7).

Листинг 6.7. Восстановление вывода в стандартный поток. Файл stdout.rb

```
$stdout = StringIO.new

puts 'Hello, world!'
File.write('output.log', $stdout.string)

$stdout = STDOUT

puts 'Hello, Ruby!'
```

Программа из листинга 6.6 перехватит фразу 'Hello, world!' и направит ее файл output.log, после этого вывод в стандартный поток восстанавливается, и фраза 'Hello, Ruby!' выводится в консоль.

Отправить фразу 'Hello, Ruby!' в стандартный поток можно без его переопределения, вызвав метод puts для объекта STDOUT (листинг 6.8).

Листинг 6.8. Вызов метода puts объекта STDOUT. Файл stdout_puts.rb

```
$stdout = StringIO.new

puts 'Hello, world!'
File.write('output.log', $stdout.string)

STDOUT.puts 'Hello, Ruby!'
```

Результат работы этой программы полностью аналогичен работе программы из листинга 6.7.

Предопределенная константа DATA позволяет извлечь данные из текущего файла после ключевого слова __END__. В программе все, что расположено после ключевого слова __END__, называется *секцией данных* (см. *разд. 2.2*). В листинге 6.9 приводится пример извлечения фразы 'Hello, world!' при помощи константы DATA.

Листинг 6.9. Извлечение секции данных при помощи константы DATA. Файл data.rb

```
puts DATA.read

__END__
Hello, world!
```

Константа DATA возвращает ссылку на поток, поэтому для извлечения данных мы используем метод read (см. *главу 27*).

6.3. Ключевые слова *__LINE__* и *__FILE__*

При помощи ключевого слова __LINE__ можно получить текущий номер строки в файле, в то время как ключевое слово __FILE__ возвращает имя файла, в которой расположена программа (листинг 6.10). Эти ключевые слова часто используются для формирования отчета в случае возникновения нештатных ситуаций и ошибок.

Листинг 6.10. Извлечение текущей строки и имени файла. Файл line_and_file.rb

```
puts __FILE__ # line_and_file.rb
puts __LINE__ # 2
```

Ключевые слова __LINE__ и __FILE__ очень похожи на константы, однако, наряду с __END__, являются именно ключевыми словами языка Ruby. Впрочем, возвращаемые ими значения являются полноценными объектами: имя файла — строкой, номер строки — целым числом.

6.4. Метод *require*

Возможность изменения значения константы очень сильно отличает язык Ruby от большинства языков программирования, где это запрещено. Дело в том, что константы в Ruby — это часть механизма импорта объектов из одного места программы в другое.

Например, классы и модули стараются размещать в отдельных файлах. Давайте создадим простейший класс HelloWorld, который будет содержать единственный метод greeting. Содержимое класса разместим в файле hello_world.rb (листинг 6.11).

Листинг 6.11. Класс `HelloWorld`. Файл hello_world.rb

```
class HelloWorld
  def greeting
    puts 'Hello, world!'
  end
end
```

Для того чтобы воспользоваться классом `HelloWorld` в другом файле, нам потребуется его подключить при помощи метода `require` (листинг 6.12). Для подключения содержимого файла hello_world.rb мы передаем его имя методу `require` в качестве аргумента.

Листинг 6.12. Использование класса `HelloWorld`. Файл hello_world_use.rb

```
require './hello_world'

hello = HelloWorld.new
hello.greeting # Hello, world!
```

В названии файла, который передается методу `require`, всегда опускается расширение. Вместо `'./hello_world.rb'` мы используем строку `'./hello_world'`. Это соглашение введено в связи с тем, что метод `require` позволяет подключать Ruby-код не только из внешних файлов, но и из скомпилированных библиотек.

Классы в Ruby — это объекты, а название `HelloWorld`, которое начинается с прописной буквы, — это константа. Метод `require` извлекает из файла только константы, игнорируя любые элементы языка, набранные строчными буквами.

Таким образом, соглашение об именовании классов и модулей в CamelCase-стиле, а локальных переменных в snake-стиле, продиктованы особенностью импорта в Ruby.

В листинге 6.13 создается файл imports.rb, в котором создаются константа `CONST` и локальная переменная `hello`.

Листинг 6.13. Файл imports.rb

```
CONST = 1
hello = 'Hello, world!'
```

В листинге 6.14 файл imports.rb подключается при помощи метода `require`, и осуществляется попытка воспользоваться константой и переменной из только что подключенного метода.

Листинг 6.14. Использование класса `HelloWorld`. Файл imports_use.rb

```
require './imports'

puts CONST # 1
puts hello # undefined local variable or method `hello'
```

Константой CONST можно успешно воспользоваться, в то время как локальная переменная hello не определена. При попытке обратиться к ней возникает ошибка.

Метод require обладает рядом особенностей — например, он загружает файл только один раз. В этом легко убедиться, выполнив его в интерактивном Ruby, где можно увидеть возвращаемый результат.

```
 > require './imports'
=> true
 > require './imports'
=> false
```

Как можно видеть, при повторной попытке метод require вернул false. Если необходимо, чтобы код из файла imports.rb выполнялся при каждом включении, лучше воспользоваться методом load:

```
 > load './imports.rb'
=> true
 > load './imports.rb'
warning: already initialized constant CONST
warning: previous definition of CONST was here
 => true
```

6.5. Метод *require_relative*

В предыдущем разделе для включения Ruby-файла из текущего каталога мы размещали перед ним ссылку на текущий каталог, который обычно обозначается точкой:

```
require './imports'
```

Связано это с тем, что инструкция require ищет файл либо по точному абсолютному пути, либо по одному из путей, указанному в глобальной переменной $LOAD_PATH:

```
require '/home/igor/imports'
```

По умолчанию ссылки на текущий каталог в массиве $LOAD_PATH нет, поэтому один из вариантов избавиться от точки в пути к файлу — добавление текущего каталога в массив $LOAD_PATH (листинг 6.15).

> ***Замечание***
>
> Оператор << добавляет в массив новый элемент. Более подробно он рассматривается в *главе 23*.

Листинг 6.15. Модификация $LOAD_PATH. Файл load_path_change.rb

```
$LOAD_PATH << '.'
require 'imports'

puts CONST # 1
```

Другой способ добиться более элегантного кода — воспользоваться методом `require_relative`, который ожидает в качестве аргумента путь относительно текущего каталога (листинг 6.16).

Листинг 6.16. Использование метода `require_relative`. Файл require_relative.rb

```
require_relative 'imports'
puts CONST # 1
```

6.6. Подключение стандартных классов

В *главе 4* были рассмотрены предопределенные классы, для которых Ruby предоставляет синтаксические конструкторы. Для использования таких классов нам не требовалось подключать их при помощи метода `require`.

Более того, в стандартную библиотеку Ruby входит большое количество других классов, которые можно использовать без подключения. Одним из них является класс `Time`, который позволяет создавать объекты, представляющие дату и время (листинг 6.17).

Листинг 6.17. Создание объекта класса `Time`. Файл mktime.rb

```
t = Time.mktime(2019, 10, 31, 21, 30, 15)
p t.to_a
```

В листинге 6.17 создается объект `t`, представляющий дату 31.10.2019 21:30:15, после чего у него вызывается метод `to_a` для получения массива с компонентами даты и времени. Вызов программы приведет к следующему результату:

```
[15, 30, 21, 31, 10, 2019, 4, 304, false, "MSK"]
```

Для извлечения года, месяца и дня класс `Time` предоставляет отдельные методы:

```
t.year  # 2019
t.month # 10
t.day   # 31
```

Помимо класса `Time`, стандартная библиотека Ruby предоставляет класс `Date`. Однако воспользоваться им без предварительного подключения не получится:

```
> Date.new
NameError (uninitialized constant Date)
```

Мы получаем здесь сообщение об ошибке: неизвестная константа `Date`. Для того чтобы воспользоваться этим классом, нам потребуется явно подключить библиотеку с ним при помощи метода `require` (листинг 6.18).

Листинг 6.18. Использование класса `Date`. Файл date.rb

```
require 'date'
p Date.new
```

Стандартная библиотека содержит множество интересных классов. Например, класс ERB позволяет не просто читать содержимое ERB-файлов, но и выполнять Ruby-выражения, не прибегая к утилите erb (см. *главу 3*). Библиотеку с классом ERB потребуется подключить при помощи метода require (листинг 6.19).

Листинг 6.19. Использование класса ERB. Файл erb.rb

```
require 'erb'

template = 'Текущее время <%= Time.now %>'
puts ERB.new(template).result
# Текущее время 2019-10-31 10:00:46 +0300
```

Таким образом, в Ruby-программе без предварительного объявления доступен лишь ограниченный набор классов. Большинство классов придется подключать при помощи метода require — даже те, что включены в стандартную библиотеку Ruby. Такое положение вещей вызвано желанием минимизировать объем памяти, которую потребляет интерпретатор Ruby. Без подключения доступны только самые распространенные и часто используемые классы.

6.7. Подключение гемов

При работе со сторонними гемами также потребуется их подключение. В *разд. 3.6.1* мы уже использовали метод require для подключения гема pry (листинг 6.20).

Листинг 6.20. Подключение гема pry. Файл pry.rb

```
require 'pry'

class HelloWorld
  def greeting
    binding.pry
    puts 'Hello, world!'
  end
end

hello = HelloWorld.new
hello.greeting
```

Если pry не установлен, выполнение команды завершится сообщением об ошибке:

```
`require': cannot load such file -- pry (LoadError)
```

В этом случае необходимо установить гем при помощи утилиты gem:

```
$ gem install pry
```

В том случае, если вы используете гем bundler для управления гемами, в конфигурационный файл Gemfile следует добавить вызов метода gem, передав ему в качестве имени строку с названием гема. В листинге 6.21 мы подключаем лишь один гем

pry, однако в крупном проекте конфигурационный файл Gemfile может подключать десятки гемов.

Листинг 6.21. Подключение гема pry через bundler. Файл pry/Gemfile

```
source 'https://rubygems.org'
gem 'pry'
```

В этом случае у нас остается возможность подключения гема pry через отдельную конструкцию require. Однако так не поступают, и вместо этого в программу подключают гем bundler, который подключает все гемы из файла Gemfile (листинг 6.22).

Листинг 6.22. Подключение bundler в проект. Файл pry/bundler.rb

```
require 'rubygems'
require 'bundler/setup'

Bundler.require(:default)

class HelloWorld
  def greeting
    binding.pry
    puts 'Hello, world!'
  end
end

hello = HelloWorld.new
hello.greeting
```

Здесь вместо отдельных require-вызовов под каждый гем вызывается один метод require класса Bundler. В качестве аргумента метод принимает название группы. Символ :default означает группу гемов из глобальной области видимости Gemfile. При помощи метода group мы можем вводить свои собственные группы. В листинге 6.23 приводится пример создания группы :debug в файле Gemfile.

> ***Замечание***
>
> Метод gem принимает блок, который формируется между ключевыми словами do и end. Более подробно блоки рассматриваются в *главе 12*.

Листинг 6.23. Создание группы :debug. Файл debug/Gemfile

```
source 'https://rubygems.org'

# Bundler.require(:default)
gem 'rubocop'

# Bundler.require(:debug)
group :debug do
  gem 'pry'
end
```

Теперь для подключения гема pry мы должны передать методу Bundler.require название группы :debug. Если необходимо подключить несколько групп — например, гемы глобальной области видимости :default и гемы группы :debug, они перечисляются в вызове Bundler.require через запятую (листинг 6.24).

Листинг 6.24. Подключение нескольких групп гемов. Файл debug/bundler.rb

```
require 'rubygems'
require 'bundler/setup'

Bundler.require(:defult, :debug)
...
```

Если нет необходимости подключать классы гема к проекту, этого можно избежать, добавив параметр require: false в вызов метода gem. Например, если мы используем гем rubocop только ради консольной команды, можно исключить подключение его кода в Gemfile:

```
gem 'rubocop', require: false
```

Задания

1. Создайте константы, которые содержат названия дней недели. Используя эти константы, сформируйте массив дней недели и присвойте его константе WEEK. Выведите содержимое массива в консоль.
2. Создайте программу number.rb, которая принимает в качестве аргумента целое число, — например: number.rb 15. В качестве результата программа должна сообщать, передано ей четное или нечетное число.
3. Создайте программу factorial.rb, которая принимает в качестве аргумента целое число, — например: factorial.rb 5. В качестве результата программа должна выводить факториал переданного числа (факториалом здесь будет произведение всех чисел от 1 до 5: 1 × 2 × 3 × 4 × 5 = 120).
4. Создайте программу sum.rb, которая принимает в качестве аргументов последовательность чисел, — например: sum.rb 6 3 7 12 2. Подсчитайте сумму чисел и выведите результат.
5. Создайте программу age.rb, которая принимает год, месяц и день рождения пользователя. Вычислите возраст пользователя и выведите результат.
6. Создайте класс Hello в файле hello.rb. Внутри класса необходимо реализовать метод hello, который выводит приветствие в зависимости от текущего времени суток. С 6:00 до 12:00 метод должен возвращать «Доброе утро», с 12:00 до 18:00 — «Добрый день», с 18:00 до 00:00 — «Добрый вечер», с 00:00 до 6:00 — «Доброй ночи». Подключите класс в файле main.rb и выведите приветствие с его помощью.

ГЛАВА 7

Операторы

Файлы с исходными кодами этой главы находятся в каталоге *operators* сопровождающего книгу электронного архива.

В предыдущих главах были затронуты несколько операторов языка Ruby — мы рассмотрели оператор присваивания, `heredoc`-оператор, арифметические операторы, оператор сравнения и оператор точки для вызова методов объектов.

В этой главе операторы будут рассмотрены более детально. Мы сосредоточимся на арифметических, поразрядных операторах, форматировании и операторе безопасного вызова. Логические операторы более детально будут рассмотрены в *главе 8*.

7.1. Операторы — это методы

Операторы — это конструкции языка, которые выполняют операцию над одним или несколькими операндами. Как правило, операнда два, однако встречаются операторы и с одним операндом — например, задание отрицательного числа при помощи оператора «минус»: `-6`. Операторы, которые ожидают один, два или три операнда называют *унарными*, *бинарными* и *тернарными* соответственно.

Простейшим примером использования операторов является операция сложения, в которой слева и справа от оператора `+` располагаются операнды:

```
> 5 + 2
=> 7
```

Однако Ruby-интерпретатор рассматривает это выражение как вызов метода `+` у объекта `5`. В качестве аргумента методу передается объект `2`:

```
> 5.+(2)
=> 7
```

В Ruby допускается не указывать круглые скобки при вызове методов:

```
> 5.+ 2
=> 7
```

Кроме того, язык Ruby позволяет не указывать перед методами-операторами точку. Поэтому вызов метода + выглядит так, как будто это обычное арифметическое сложение.

7.2. Арифметические операторы

Ruby поддерживает стандартные арифметические операторы, которые представлены в табл. 7.1.

Таблица 7.1. *Арифметические операторы*

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
**	Возведение в степень

Операторы сложения, вычитания, умножения и деления используются по правилам, принятым в арифметике: сначала выполняются операторы умножения и деления и лишь затем операторы сложения и вычитания. Для того чтобы изменить такой порядок, следует прибегать к группированию чисел при помощи круглых скобок (см. листинг 5.7).

> ***Замечание***
>
> Традиционно до и после арифметического оператора помещаются пробелы, т. к. это позволяет увеличить читабельность программы. Однако это необязательное требование — так, выражение в листинге 7.1 может быть записано следующим образом: `(5+3)*(6+7)`.

Листинг 7.1. Использование арифметических операторов. Файл arithmetic.rb

```
puts (5 + 3) * (6 + 7) # 104
```

При делении целых чисел Ruby отбрасывает целую часть. Получить ее можно при помощи оператора взятия остатка от деления (листинг 7.2).

Листинг 7.2. Деление целых чисел. Файл division.rb

```
puts 11 / 3 # 3
puts 11 % 3 # 2
```

Если при делении вместо целых чисел хотя бы один из операндов будет вещественным, мы получим более точный результат:

```
 > 11 / 3.0
=> 3.6666666666666665
```

При целочисленном делении дробная часть отбрасывается так, как будто в числителе вместо `11` подставлено значение `9`. Оператор взятия остатка деления возвращает разницу между этими цифрами: `11 - 9 = 2`.

Если в качестве делителя используется число `2`, по остатку деления можно определять, четное перед нами число или нечетное. Для четных чисел остаток от деления всегда возвращает `0`, для нечетных — `1` (листинг 7.3).

Листинг 7.3. Определение четности и нечетности числа. Файл even_or_odd.rb

```
puts 23 % 2 # 1 - нечетное
puts 26 % 2 # 0 - четное
```

Впрочем, в этом нет необходимости, — для целых чисел предусмотрены специальные методы определения четности и нечетности числа (листинг 7.4).

Листинг 7.4. Использование методов even? и odd?. Файл even_or_odd_methods.rb

```
p 23.even? # false - число не является четным
p 23.odd?  # true  - число нечетное

p 26.even? # true  - число четное
p 26.odd?  # false - число не является нечетным
```

7.3. Присваивание

В *разд. 5.2* уже рассматривался оператор присваивания =, позволяющий связывать переменные и объекты:

```
number = 1
str = 'Hello, world!'
```

Этот оператор обладает несколькими дополнительными формами и свойствами, которые рассматриваются в следующих разделах.

7.3.1. Сокращенная форма арифметических операторов

Помимо операторов, приведенных в табл. 7.1, можно также использовать сокращенную запись арифметических операторов (табл. 7.2).

Пусть имеется операция сложения переменной `var` с последующим присваиванием результата этой же самой переменной:

```
 > var = 0
=> 0
 > var = var + 10
=> 10
```

Таблица 7.2. Сокращенные операторы

Сокращенная форма	Полная форма
`var += 10`	`var = var + 10`
`var -= 10`	`var = var - 10`
`var *= 10`	`var = var * 10`
`var /= 10`	`var = var / 10`
`var %= 10`	`var = var % 10`
`var **= 10`	`var = var ** 10`

Последнее выражение можно записать в сокращенной форме:

```
 > var += 10
=> 10
```

Такая сокращенная форма допустима для всех арифметических операторов.

> ***Замечание***
> В Ruby отсутствуют операторы инкремента ++ и декремента --.

7.3.2. Параллельное присваивание

До текущего момента при помощи оператора присваивания одной переменной сопоставлялось одно значение. В Ruby допускается параллельное присваивание, когда слева и справа от оператора «равно» располагаются несколько переменных и значений (листинг 7.5).

Листинг 7.5. Параллельное присваивание. Файл parallel_assign.rb

```
fst, snd, thd = 'Hello', 'world', '!'
puts fst # Hello
puts snd # world
puts thd # !
```

В случае параллельного присваивания переменные и значения перечисляются через запятую. При помощи этого приема можно поменять местами значения переменных, не прибегая к использованию третьей переменной (листинг 7.6).

Листинг 7.6. Обмен значений переменных. Файл var_change.rb

```
fst = 'первый'
snd = 'второй'
fst, snd = snd, fst
p fst # "второй"
p snd # "первый"
```

Слева и справа от оператора «равно» может быть неодинаковое количество элементов. В этом случае переменные слева, которым не хватило значения справа, получают значение `nil` (листинг 7.7).

Листинг 7.7. Файл parallel_assign_nil.rb

```
fst, snd, thd = 'Hello, world!'
p fst # Hello, world!
p snd # nil
p thd # nil
```

Ситуация с одним значением справа меняется, если оно может быть неявно преобразовано в список. В листинге 7.8 массив `['Hello', 'world', '!']` рассматривается как список.

Листинг 7.8. Файл parallel_assign_array_split.rb

```
fst, snd, thd = ['Hello', 'world', '!']
puts fst # Hello
puts snd # world
puts thd # !
```

В случае, если слева от оператора «равно» лишь одна переменная, а справа перечислено несколько значений через запятую, — они упаковываются в массив (листинг 7.9).

Листинг 7.9. Файл parallel_assign_array.rb

```
arr = 'Hello', 'world', '!'
p arr # ["Hello", "world", "!"]
```

Если переменных слева несколько, но их меньше, чем значений справа, лишние значения справа отбрасываются (листинг 7.10).

Листинг 7.10. Файл parallel_assign_ignore.rb

```
fst, snd = 'Hello', 'world', '!'
puts fst # Hello
puts snd # world
```

7.3.3. Круглые скобки в параллельном присваивании

При параллельном присваивании элементы, заключенные в круглые скобки, ведут себя особым образом — содержимое круглых скобок рассматривается как один элемент (листинг 7.11).

Листинг 7.11. Использование круглых скобок. Файл parallel_assign_parentheses.rb

```
fst, (f, s), thd = 'Hello', 'world', '!'
p fst # "Hello"
p f   # "world"
p s   # nil
p thd # "!"
```

В листинге 7.11 содержимое круглых скобок `(f, s)` соответствует элементу `'world'`. То есть присваивание с участием круглых скобок можно рассматривать следующим образом:

```
f, s = 'world'
```

Если вместо строки `'world'` в листинге 7.11 использовать массив, можно заполнить элемент `s` (листинг 7.12).

Листинг 7.12. Файл parallel_assign_array_parentheses.rb

```
fst, (f, s), thd = 'Hello', ['world', 'Ruby'], '!'
p fst # "Hello"
p f   # "world"
p s   # "Ruby"
p thd # "!"
```

7.3.4. Оператор *

Оператор `*` позволяет разложить элементы массива на составляющие. В листинге 7.13 приводится пример параллельного присваивания с использованием и без использования оператора `*`.

Листинг 7.13. Использование оператора *. Файл splat_operator.rb

```
fst, snd, thd = 'Hello', ['world', '!']
p fst # "Hello"
p snd # ["world", "!"]
p thd # nil

fst, snd, thd = 'Hello', *['world', '!']
p fst # "Hello"
p snd # "world"
p thd # "!"
```

В первом случае переменная `snd` получает в качестве значения массив, а третья переменная `thd` — значение `nil`. При использовании оператора `*` массив раскладывается на два элемента: переменная `snd` получает в качестве значения первый элемент, а переменная `thd` — второй.

Оператор * можно использовать слева от оператора равенства. В случае, если значений справа будет больше, чем значений слева, переменная, около которой указан оператор *, будет получать в качестве значения массив (листинг 7.14).

Листинг 7.14. Файл splat_operator_var.rb

```
fst, *snd = ['Hello', 'world', '!']
p fst # "Hello"
p snd # ["world", "!"]

*fst, snd = ['Hello', 'world', '!']
p fst # ["Hello", "world"]
p snd # "!"
```

В том случае, если переменной с оператором * не будет хватать элементов, вместо значения nil она получает пустой массив (листинг 7.15).

Листинг 7.15. Файл splat_operator_empty_array.rb

```
fst, snd, thd, *fth = ['Hello', 'world', '!']
p fst # "Hello"
p snd # "world"
p thd # "!"
p fth # []
```

Оператор * может одновременно появляться слева и справа от знака равенства (листинг 7.16).

Листинг 7.16. Файл splat_operator_both.rb

```
fst, *snd = *['Hello', 'world', '!']
p fst # "Hello"
p snd # ["world", "!"]
```

Раскладывать массив в список можно не только в параллельном присваивании. Например, методы puts и p могут принимать неограниченное количество элементов. Если передать методу p массив — он будет выведен в одну строку. Если разложить массив при помощи оператора *, каждый элемент массива будет выведен на отдельной строке (листинг 7.17).

Листинг 7.17. Разложения массива в список аргументов. Файл splat_operator_array.rb

```
p [1, 2, 3, 4, 5]
p *[1, 2, 3, 4, 5]
```

Результатом выполнения программы из листинга 7.17 будут следующие строки:

```
[1, 2, 3, 4, 5]
1
2
3
4
5
```

Оператор * может применяться не только в отношении массивов. Его можно использовать совместно с диапазонами. В листинге 7.18 для создания массива из 10 элементов от 1 до 10 можно использовать разложение диапазона 1..10, передав его синтаксическому конструктору [].

Листинг 7.18. Создание массива. Файл splat_operator_range.rb

```
p [*1..10] # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

В диапазонах могут использоваться не только цифры. Прием из листинга 7.18 можно применить для создания массива букв русского алфавита. Для этого можно воспользоваться диапазоном 'а'..'я':

```
 > [*'а'..'я']
=> ["а", "б", "в", "г", ..., "ы", "ь", "э", "ю", "я"]
```

7.4. Операторы строк

Мы уже использовали оператор [] для доступа к отдельным элементам строки и подстрокам (см. *главу 4*). Это не единственный оператор, который может использоваться совместно со строками. Например, для добавления подстроки к строке можно применить оператор << (листинг 7.19).

Листинг 7.19. Добавление подстроки в строку. Файл string_append.rb

```
str = 'Hello,'
str << ' world!'
puts str # Hello, world!
```

В следующих разделах мы рассмотрим и другие операторы, которые могут быть использованы со строками.

7.4.1. Умножение строки на число

Многие из операторов, представленных в табл. 7.1, можно применять не только к числовым значениям, но и к другим объектам языка Ruby. Так как оператор это метод, необходимо, чтобы он был реализован на уровне класса объекта.

Например, при умножении строки на число можно получить новую строку, в которой исходная строка повторяется столько раз, насколько она умножается (листинг 7.20).

Листинг 7.20. Сложение строк. Файл string_multi.rb

```
puts 'Hello' * 3 # HelloHelloHello
```

В случае, если строка связана с переменной, допускается использование сокращенной формы оператора присваивания (листинг 7.21).

Листинг 7.21. Сокращенная форма оператора *=. Файл string_multi_short.rb

```
str = 'Hello'
str *= 3
puts str # HelloHelloHello
```

7.4.2. Сложение строк

Сложение строк можно осуществить двумя способами: либо с использованием оператора +, либо при помощи метода concat. В обоих случаях получается один и тот же результат (листинг 7.22).

Листинг 7.22. Сложение строк. Файл string_concat.rb

```
puts 'Hello,' + ' world!'       # Hello, world!
puts 'Hello,'.concat(' world!') # Hello, world!
```

Однако, если мы попробуем сложить строки с числами, нас ждет неудача:

```
> 'Стоимость ' + 500 + ' рублей'
TypeError (no implicit conversion of Integer into String)
```

Обойти эту ситуацию можно при помощи метода to_s, который преобразует число 500 в строку:

```
 > 'Стоимость ' + 500.to_s + ' рублей'
=> "Стоимость 500 рублей"
```

Помимо сложения, для формирования конечной строки можно воспользоваться механизмом интерполяции (листинг 7.23).

Листинг 7.23. Интерполяция числа в строку. Файл string_interpolate.rb

```
puts "Стоимость #{500} рублей" # Стоимость 500 рублей
```

Интерполяция начинается с символа решетки, после которого следуют фигурные скобки, в которых можно разместить любое Ruby-выражение. Результат вычисления этого выражения будет вставлен в строку:

```
 > "Возведем два в степень восемь: #{ 2 ** 8 }"
=> "Возведем два в степень восемь: 256"
```

Интерполяция работает за счет неявного вызова метода `to_s`. Пример из листинга 7.13 для Ruby-интерпретатора выглядит следующим образом:

```
puts "Стоимость #{500.to_s} рублей" # Стоимость 500 рублей
```

Так как метод `to_s` вызывается неявно, то в случае интерполяции он всегда опускается.

7.4.3. Форматирование строк

Интерполяция не единственный способ вставки числа в строку. Для этого также можно воспользоваться оператором `%`. Внутри строки, помимо обычных символов, могут содержаться начинающиеся со знака `%` специальные последовательности символов, которые называют *определителями преобразования*. В примере, представленном в листинге 7.24, определитель преобразования `%d` подставляет в строку число, которое передается в качестве второго аргумента функции.

Листинг 7.24. Использование оператора `%`. Файл format_operator.rb

```
puts "Стоимость %d рублей" % 500 # Стоимость 500 рублей
```

Буква `d`, следующая за знаком `%`, определяет тип аргумента (целое, строка и т. д.), поэтому называется *определителем типа*. В табл. 7.3 представлены определители типа, которые допускаются в строке при использовании оператора `%`.

***Таблица 7.3.** Определители типа*

Определитель	Описание
`%a`	Определитель вещественного числа для подстановки в строку в шестнадцатеричном формате
`%A`	Аналогичен `%a`, однако префикс `0X` и экспонента `P` используют прописные, а не строчные буквы
`%b`	Определитель целого, которое выводится в виде двоичного числа. При использовании альтернативного синтаксиса `%#b` двоичное число предваряется префиксом `0b`
`%B`	Аналогичен `%b`, однако при использовании альтернативного синтаксиса `%#B` в префиксе используется прописная буква `0B`
`%c`	Спецификатор символа строки используется для подстановки в строку формата одного символа — например: `'a'`, `'w'`, `'0'`, `'\0'`
`%d`	Спецификатор десятичного целого числа используется для подстановки целых чисел — например: 0, 100, –45. Допускается использование синонимов `%i` и `%u`
`%e`	Спецификатор числа в экспоненциальной нотации — например, число 1200 в этой нотации записывается как `1.2e+03`, а 0.01, как `1e-02`
`%E`	Аналогичен `%e`, однако для обозначения экспоненциальной нотации используется прописная буква `E`: `1.2E+03` и `1E-02`

Таблица 7.3 *(окончание)*

Определитель	Описание
`%f`	Спецификатор вещественного числа — например, 156.001
`%g`	Спецификатор десятичного числа с плавающей точкой, который ведет себя как `%f`, однако для чисел меньше 10^{-4} (0.00001) ведет себя как спецификатор экспоненциальной нотации `%e`
`%G`	Аналогичен `%g`, однако для обозначения экспоненциальной нотации используется прописная буква `E`
`%o`	Спецификатор для подстановки в строку формата восьмеричного числа без знака
`%p`	Вызывает для вставляемого значения `var` метод `var.inspect` и подставляет результат в строку
`%s`	Спецификатор для подстановки в строку формата строки
`%x`	Спецификатор для подстановки в строку формата шестнадцатеричного числа (строчные буквы для `a`, `b`, `c`, `d`, `e`, `f`)
`%X`	Спецификатор для подстановки в строку формата шестнадцатеричного числа (прописные буквы для `A`, `D`, `C`, `D`, `E`, `F`)
`%%`	Обозначение одиночного символа `%` в строке вывода

В листинге 7.25 демонстрируется форматный вывод числа `5867` с использованием разнообразных определителей типа.

Листинг 7.25. Работа с определителями типа. Файл format_integer.rb

```
number = 5867

puts '%b' % number  # 1011011101011
puts '%b' % -number # ..10100100010101
puts '%#b' % number # 0b1011011101011
puts '%#B' % number # 0B1011011101011
puts '%d' % number  # 5867
puts '%f' % number  # 5867.000000
puts '%o' % number  # 13353
puts '%o' % -number # ..764425
puts '%#o' % number # 013353
puts '%s' % number  # 5867
puts '%x' % number  # 16eb
puts '%x' % -number # ..fe915
puts '%X' % number  # 16EB
```

Обратите внимание, что в случае отрицательных чисел перед ними выводятся две точки.

В строке допускается использование сразу нескольких определителей. В листинге 7.26 при помощи трех определителей `%X` формируется цвет в HTML-нотации. В этом случае слева от оператора `%` ожидается массив аргументов.

Листинг 7.26. Использование нескольких определителей. Файл format_color.rb

```
red = 255;
green = 255;
blue = 100;
puts '#%X%X%X' % [red, green, blue] # #FFFF64

color = [255, 255, 100]
puts '#%X%X%X' % color # #FFFF64
```

Между символом `%` и определителем типа может быть расположен *определитель заполнения*. Определитель заполнения состоит из символа заполнения и числа, которое определяет, сколько символов отводится под вывод. Все не занятые параметром символы будут заполнены символом заполнителя. Так, в листинге 7.27 под вывод числа `45` отводятся пять символов, но поскольку само число занимает лишь два символа, три ведущих символа будут содержать символ заполнения.

Листинг 7.27. Форматирование целого числа. Файл format_integer_number.rb

```
p '% 4d' % 45 # "   45"
p '%04d' % 45 # "00045"
```

Применение определителя типа `f` позволяет вывести число в десятичном формате. Очень часто требуется вывести строго определенное число символов после запятой — например, для вывода денежных единиц, где обязательным требованием являются два знака после запятой. В этом случае прибегают к *определителю точности*, который следует сразу за определителем ширины и представляет собой точку и число символов, отводимых под дробную часть числа (листинг 7.28).

Листинг 7.28. Форматирование вещественного числа. Файл format_float.rb

```
p '%8.2f' % 1000.45684 # " 1000.46"
p '%.2f' % 12.92869    # "12.93"
```

В первом случае под все число отводятся 8 символов, два из которых будут заняты мантиссой числа. Во втором случае ограничение накладывается только на количество цифр после точки.

Как видно из листинга 7.28, цифры округляются. Для этой операции предназначен специальный метод `round`, который может принимать в качестве необязательного параметра количество знаков после запятой, до которого необходимо округлить число (листинг 7.29).

Листинг 7.29. Округление числа. Файл round.rb

```
puts 1000.45684.round    # 1000
puts 1000.45684.round(2) # 1000.46
```

```
puts 12.92869.round      # 13
puts 12.92869.round(2)   # 12.93

puts 1000.45684.ceil    # 1001
puts 1000.45684.ceil(2) # 1000.46
puts 12.92869.ceil      # 13
puts 12.92869.ceil(2)   # 12.93

puts 1000.45684.floor    # 1000
puts 1000.45684.floor(2) # 1000.45
puts 12.92869.floor      # 12
puts 12.92869.floor(2)   # 12.92
```

Метод `round` производит округление по математическим правилам: цифры до пяти округляются в меньшую сторону, большие или равные пяти — в большую. Поэтому при округлении до второго знака после запятой число 1000.45684 округляется в большую сторону — до 1000.46, а при округлении до целого числа — в меньшую до 1000.

Помимо `round`, к числам можно применять дополнительные методы: `ceil` и `floor`. Метод `ceil` производит округление до ближайшего целого числа, причем результат всегда *больше* округляемого числа. Метод `floor` производит округление до ближайшего целого числа, причем результат всегда *меньше* округляемого числа (рис. 7.1).

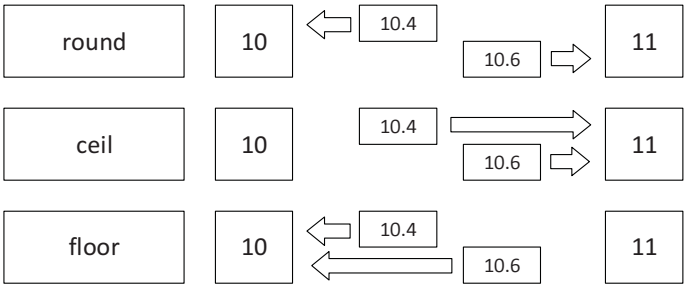


Рис. 7.1. Округление числа при помощи методов `round`, `ceil` и `floor`

Оператор `%` является сокращением для двух методов: `sprintf` и `format`, которые позволяют выполнять все описанные ранее операции (листинг 7.30).

Листинг 7.30. Использование методов `sprintf` и `format`. Файл format.rb

```
puts "Стоимость %d рублей" % 500         # Стоимость 500 рублей
puts sprintf("Стоимость %d рублей", 500) # Стоимость 500 рублей
puts format("Стоимость %d рублей", 500)  # Стоимость 500 рублей
```

Все три варианта в листинге 7.30 являются полностью эквивалентными. Не рекомендуется злоупотреблять оператором `%` — считается, что метод `format` более на-

гляден и читаем. Начиная с этого момента, в книге мы будем использовать именно метод `format`.

Иногда один и тот же параметр может несколько раз присутствовать в строке форматирования. В этом случае допускается использовать числовые индексированные параметры, которые в строке обозначаются при помощи символа доллара: `1$` — первый параметр, `2$` — второй и т. д. (листинг 7.31).

Листинг 7.31. Использование индексированных параметров. Файл format_index.rb

```
puts format('%d + %d = %d', 2, 2, 2 + 2)    # 2 + 2 = 4
puts format('%1$d + %1$d = %2$d', 2, 2 + 2) # 2 + 2 = 4
```

В листинге 7.31 представлены две строки, которые приводят к одному и тому же результату — выводят выражение: `'2 + 2 = 4'`. В первом случае мы вынуждены слагаемое `2` повторять два раза, во-втором — только один. Достигается это размещением между символами `%` и `d` номера аргумента `1$`. На значение `2 + 2` в списке аргументов можно сослаться при помощи индекса `2$`. В случае, если среди определителей много повторяющихся, за счет индексированных параметров можно значительно сократить код.

Вместо числовых индексов для обозначения определителей можно использовать именованные параметры. Для этого между `%` и определителем размещается название параметра в угловых скобках (листинг 7.32).

> ***Замечание***
>
> Именованные параметры в вызове метода `format` на самом деле представляют собой хэш `format('...', {summand: 2, result: 2 + 2})`. При вызовах методов у аргументов-хэшей, расположенных последними, можно и нужно не указывать фигурные скобки. Более подробно методы и параметры рассматриваются в *главе 9*.

Листинг 7.32. Использование именованных параметров. Файл format_name.rb

```
puts format(
  '%<summand>d + %<summand>d = %<result>d',
  summand: 2,
  result: 2 + 2
)
```

Для подстановки значений в именованные определители используются символы, которые совпадают с именами определителей. Каждому из символов сопоставляется значение, которое необходимо подставить вместо него.

7.5. Операторы сравнения

Язык программирования Ruby предоставляет большой набор операторов сравнения (табл. 7.4). Все операторы, за исключением `<=>`, возвращают логическое значение: либо `true`, либо `false`.

Таблица 7.4. *Операторы сравнения*

Оператор	Описание
>	Оператор «больше» возвращает `true`, если левый операнд больше правого
<	Оператор «меньше» возвращает `true`, если левый операнд меньше правого
>=	Оператор «больше равно» возвращает `true`, если левый операнд больше или равен правому операнду
<=	Оператор «меньше равно» возвращает `true`, если левый операнд меньше или равен правому операнду
==	Оператор равенства возвращает `true`, если сравниваемые операнды равны
!=	Оператор неравенства возвращает `true`, если сравниваемые операнды не равны
<=>	Возвращает `-1`, если левый операнд меньше правого, `0` — в случае, если операнды равны, и `1`, если левый операнд больше правого
=~	Возвращает `true`, если операнд соответствует регулярному выражению (см. *главу 30*)
!~	Возвращает `true`, если операнд не соответствует регулярному выражению (см. *главу 30*)
===	Оператор равенства, предназначенный для перегрузки в классах

В большинстве случаев операторы сравнения применяются для построения логических выражений, которые затем используются либо в операциях ветвления (см. *главу 8*), либо в циклах (см. *главу 10*). В листинге 7.33 приводится пример сравнения целых чисел друг с другом.

Листинг 7.33. Сравнение целых чисел. Файл compare_integers.rb

```
p 3 > 10   # false
p 3 < 10   # true
p 3 <=> 10 # -1
p 3 <=> 3  # 0
p 10 <=> 3 # 1
```

7.5.1. Особенности сравнения объектов

Важно отметить, что при сравнении объектов они считаются равными в том случае, если это один и тот же объект (листинг 7.34).

Листинг 7.34. Сравнение объектов. Файл compare_objects.rb

```
fst = Object.new
snd = Object.new
thd = fst

p fst == snd # false
p fst == thd # true
```

Из этого правила имеются следствия — например, если мы сравниваем диапазон `1..10` с числом `5`, мы получаем `false` (ложь).

```
 > (1..10) == 5
=> false
```

Объект диапазона `1..10` и объект числа `5` — это два разных объекта. Поэтому при сравнении с использованием оператора равенства мы получаем `false`.

Для того чтобы осуществлять сравнение по особым правилам, используется оператор `===`, поведение которого для большинства классов переопределено. Например, сравнение диапазона и объекта возвращает `true`, если объект входит в диапазон, и `false` — если нет (листинг 7.35).

Листинг 7.35. Сравнение объектов. Файл compare_range.rb

```
p (1..10) === 5              # true
p (1..10) === 11             # false
p String  ==  'Hello, world!' # false
p String  === 'Hello, world!' # true
p /\d+/   ==  '12345'        # false
p /\d+/   === '12345'        # true
```

Оператор `===` переопределен для всех классов таким образом, что возвращает `true`, если объект слева принадлежит этому классу, и `false` — в противном случае. Для регулярных выражений оператор `===` переопределен таким образом, чтобы возвращалось `true`, если операнд справа соответствует регулярному выражению, и `false` — в противном случае. Можно самостоятельно переопределять поведение операторов (см. *главу 15*).

При использовании операторов следует помнить, что в Ruby они являются методами, поэтому возвращаемый результат не всегда симметричен:

```
 > (1..10) === 5
=> true
 > 5 === (1..10)
=> false
```

Если мысленно добавить в оператор вызова метода точку и скобки — все встает на свои места:

```
 > (1..10).===(5)
=> true
 > 5.===(1..10)
=> false
```

Для класса `Range` поведение `===` переопределено, а для класса `Integer` — нет.

7.5.2. Сравнение с нулем

Весьма часто осуществляется сравнение с нулевым значением. Это может быть как прямое сравнение с участием оператора сравнения:

```
 > var = 1
=> 1
 > var == 0
=> false
 > var - 1 == 0
=> true
```

так и проверка, является ли число положительным или отрицательным:

```
 > var = 1
=> 1
 > var > 0
=> true
 > var < 0
```

Для таких операций в Ruby предусмотрено три отдельных метода (рис. 7.2):

- `negative?` — возвращает `true`, если число меньше нуля, и `false` — в противном случае;
- `zero?` — возвращает `true`, если число является нулем, и `false` — в противном случае;
- `positive?` — возвращает `true`, если число больше нуля, в противном случае возвращается `false`.

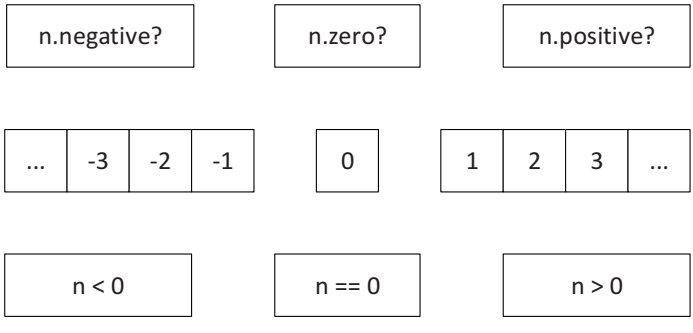


Рис. 7.2. Методы `negative?`, `zero?` и `positive?`

Специальные методы `negative?`, `zero?` и `positive?` считаются более наглядными по сравнению с операторами сравнения. Поэтому при операциях сравнения с нулем лучше использовать их (листинг 7.36).

Листинг 7.36. Сравнение с нулем. Файл compare_zero.rb

```
p 1.negative?  # false
p 0.negative?  # false
p -1.negative? # true
```

```
p 1.zero?      # false
p 0.zero?      # true
p -1.zero?     # false

p 1.positive?  # true
p 0.positive?  # false
p -1.positive? # false
```

7.5.3. Особенности сравнения вещественных чисел

Не все вещественные числа можно представить в двоичной системе счисления. За счет того, что такие числа при операциях с ними мы представляем приближенно, может накапливаться ошибка вычисления (листинг 7.37).

Листинг 7.37. Сравнение вещественных чисел. Файл compare_float.rb

```
p 1 / 3.0                 # 0.3333333333333333
p 4 / 3.0                 # 1.3333333333333333
p 4 / 3.0 - 1             # 0.33333333333333326
p 4 / 3.0 - 1 == 1 / 3.0 # false
```

Математически числа `4 / 3 - 1` и `1 / 3` должны быть эквивалентны. Однако за счет накапливающейся ошибки вычисления прямое сравнение этих двух чисел завершается неудачей.

Для решения этой задачи в классе `Float` определена константа `EPSILON`, задающая порог для погрешности вычисления. Сравниваемые величины вычитают друг из друга, и от результата берется модуль при помощи метода `abs`. Далее проверяется, является ли полученный результат меньше величины `Float::EPSILON`. Если это так, величины считаются равными друг другу, в противном случае — разными (листинг 7.38).

Листинг 7.38. Использование константы `Float::EPSILON`. Файл compare_epsilon.rb

```
fst = 4 / 3.0 - 1
snd = 1 / 3.0

p fst == snd                        # false
p (fst - snd).abs < Float::EPSILON # true
```

7.5.4. Особенности сравнения строк

Сравнение чисел и строк отличается друг от друга. Дело в том, что наши строки читаются слева направо, в то время как разряды арабских цифр отсчитываются справа налево (рис. 7.3).

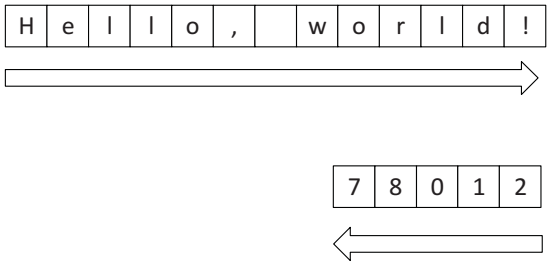


Рис. 7.3. Порядок сравнения символов в строках и разрядов в цифрах

При сравнении двух строк последовательно сравниваются коды отдельных символов. Получить их можно при помощи метода `ord`:

```
 > 'a'.ord
=> 97
 > 'b'.ord
=> 98
 > 'а'.ord
=> 1072
 > 'б'.ord
=> 1073
```

В приведенном примере были получены коды английских символов `a` и `b`, а также русских `а` и `б`. Английский и русский алфавиты закодированы таким образом, чтобы коды букв возрастали от `a` до `b` и от `а` до `я`.

```
 > 'a' > 'b'
=> false
 > 'a' < 'b'
=> true
```

Сравнение строк осуществляется слева направо, как только на очередной позиции встречаются различающиеся символы. В приведенном далее примере первые символы, которые различаются в строках, это `r` и `w` в словах `ruby` и `world`:

```
 > 'Hello, ruby!' > 'Hello, world!'
=> false
 > 'Hello, ruby!' < 'Hello, world!'
=> true
```

Код символа `w` больше, чем код символа `r`, поэтому строка `'Hello, world!'` оказывается больше строки `'Hello, ruby!'` (рис. 7.4).

Рис. 7.4. Сравнение строк

В том случае, если при сравнении строк одна из строк заканчивается раньше, чем встречаются различающиеся символы, большей считается строка, в которой больше символов:

```
 > 'Hello' < 'Hello, world!'
=> true
```

7.6. Поразрядные операторы

Эта группа операторов предназначена для выполнения операций над отдельными битами числа (табл. 7.5).

Таблица 7.5. *Поразрядные операторы*

Оператор	Описание
`&`	Поразрядное (побитовое) пересечение — И (AND)
`|`	Поразрядное (побитовое) объединение — ИЛИ (OR)
`^`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)
`~`	Поразрядное отрицание (NOT)
`<<`	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда
`>>`	Сдвиг вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда

Оператор `&` соответствует побитовому И (пересечение). При использовании битовых операторов следует помнить, что все операции выполняются над числами в двоичном представлении. Пусть имеются два числа: `6` и `10`, применение к ним оператора `&` дает `2` (листинг 7.39).

Листинг 7.39. Использование оператора &. Файл bits_and.rb

```
puts 6 & 10 # 2
```

Этот, казалось бы, парадоксальный результат можно легко объяснить, если перевести цифры `6` и `10` в двоичное представление, в котором они равны `0110` и `1010` соответственно. Сложение разрядов при использовании оператора `&` происходит согласно правилам, представленным в табл. 7.6. Учитывая их, можно легко получить результат, который равен `0010` (в десятичной системе — `2`):

```
0110 (6)
1010 (10)
0010 (2)
```

В листинге 7.40 приведен более сложный пример — использование оператора `&` для чисел `113` и `45`.

Таблица 7.6. *Правила сложения разрядов при использовании оператора* `&`

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	0	0	0

Листинг 7.40. Использование оператора `&`. Файл bits_and_use.rb

```
puts 113 & 45 # 33
```

Для наглядности представим складываемые числа в двоичном представлении:

```
1110001 (113)
0101101 (45)
0100001 (33)
```

Оператор `|` соответствует побитовому ИЛИ (объединение). В табл. 7.7 приведены правила сложения разрядов при использовании оператора `|`.

Таблица 7.7. *Правила сложения разрядов при использовании оператора* `|`

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	1	1	1	0

Применяя операцию `|` к числам, рассмотренным в предыдущем разделе, получим следующий результат (листинг 7.41).

Листинг 7.41. Использование оператора `|`. Файл bits_or.rb

```
puts 6 | 10   # 14
puts 113 | 45 # 125
```

Распишем числа в двоичном представлении, чтобы действие оператора было более понятно:

```
0110 (6)
1010 (10)
1110 (14)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1111101 (125)
```

Оператор ^ соответствует побитовому ИСКЛЮЧАЮЩЕМУ ИЛИ (XOR). В табл. 7.8 приведены правила сложения разрядов при использовании оператора ^.

Таблица 7.8. *Правила сложения разрядов при использовании оператора* ^

Операнд/результат	Значение			
Первый операнд	1	1	0	0
Второй операнд	1	0	1	0
Результат	0	1	1	0

В листинге 7.42 приведен пример использования этого оператора применительно к числам 6, 10 и 113, 45.

Листинг 7.42. Использование оператора ^. Файл bits_xor.rb

```
puts 6 ^ 10   # 12
puts 113 ^ 45 # 92
```

Распишем числа в двоичном представлении, чтобы действие оператора было более понятно:

```
0110 (6)
1010 (10)
1100 (12)
```

Для второй пары чисел:

```
1110001 (113)
0101101 (45)
1011100 (92)
```

Оператор ~ выполняет побитовую инверсию (отрицание) и заменяет все 0 на 1 и наоборот. В листинге 7.43 демонстрируется использование этого оператора.

Листинг 7.43. Использование оператора ~. Файл bits_not.rb

```
puts ~45 # -46
puts ~92 # -93
```

Все побитовые операции производятся над целыми числами, знак в них кодируется первым разрядом: если он равен 0, число положительное, если он равен 1 — отрицательное. В двоичных числах ведущие нули обычно не указывают, но поскольку операция ~ производит их инвертирование — они заменяются единицами, приводя к отрицательным значениям. Так, операция ~45 в двоичном представлении выглядит следующим образом:

```
00000000000000000000000000101101 (45)
11111111111111111111111111010010 (-45)
```

Для `~92`:

```
00000000000000000000000001011100 (92)
11111111111111111111111110100011 (-92)
```

Оператор << сдвигает влево все биты первого операнда на число позиций, указанных во втором операнде (листинг 7.44). Сдвиг на отрицательное значение приводит к нулевому результату.

Листинг 7.44. Использование оператора <<. Файл bits_left_shift.rb

```
puts 6 << 1  # 12
puts 6 << 2  # 24
puts 6 << 10 # 6144
```

В двоичном представлении эти операции выглядят следующим образом:

- `6 << 1`:

  ```
  0110 (6)
  1100 (12)
  ```

- `6 << 2`:

  ```
  00110 (6)
  11000 (24)
  ```

- `6 << 10`:

  ```
  0000000000110 (6)
  1100000000000 (6144)
  ```

Оператор >> сдвигает вправо все биты первого операнда на число позиций, указанных во втором операнде. Сдвиг на отрицательное значение приводит к нулевому результату (листинг 7.45).

Листинг 7.45. Использование оператора >>. Файл bits_right_shift.rb

```
puts 6144 >> 10 # 6
puts 45 >> 2    # 11
puts 45 > 2     # true
```

В двоичном представлении эти операции выглядят следующим образом:

- `6144 >> 10`:

  ```
  1100000000000 (6144)
  0000000000110 (6)
  ```

- `45 >> 2`:

  ```
  101101 (45)
  001011 (11)
  ```

Как видно из листинга 7.45, при использовании операторов сдвига очень легко ошибиться, указав только один знак > (см. *разд. 7.5*). В этом случае интерпретатор

Ruby вернет либо `true`, либо `false`. За такими опечатками следует следить очень внимательно.

Так же, как и арифметические операторы, поразрядные операторы поддерживают сокращенную запись, позволяющую сократить выражения вида `number = number & var` до `number &= var`. В табл. 7.9 приводятся сокращенные формы побитовых операторов.

Таблица 7.9. *Сокращенные операторы*

Сокращенная запись	Полная запись
`number &= var;`	`number = number & var;`
`number |= var;`	`number = number | var;`
`number ^= var;`	`number = number ^ var;`
`number <<= var;`	`number = number << var;`
`number >>= var;`	`number = number >> var;`

7.7. Оператор безопасного вызова

Оператор безопасного вызова `&.` введен в Ruby, начиная с версии 2.3. Оператор подавляет возникновение ошибки при попытке вызова несуществующего метода в отношении неопределенного значения `nil`.

Мы уже упоминали, что выражения в Ruby возвращают значения. В качестве значения выступают объекты, у которых можно вызывать методы. Методы, в свою очередь, также возвращают объекты, у которых снова можно вызывать метод:

```
 > hello = 'Hello, world!'
=> "Hello, world!"
 > hello.gsub('hello', 'ruby').gsub('!', '')
=> "Hello, world"
```

Таким образом может выстраиваться целая цепочка вызовов. В приведенном примере таких вызовов только два, однако на практике их может быть гораздо больше.

Проблемы могут возникнуть, если на одном из этапов будет возвращено неопределенное значение `nil`. В этом случае цепочка прерывается ошибкой:

```
 > hello = 'Hello, world!'
=> "Hello, world!"
 > hello.index('world')
=> 7
 > hello.index('world').odd?
=> true
 > hello.index('ruby').odd?
NoMethodError (undefined method `odd?' for nil:NilClass)
```

В приведенном примере мы определяем, является ли номер позиции, с которой начинается вхождение подстроки в строку, нечетным. Все работает корректно, пока метод `index` может обнаружить подстроку и вернуть числовую позицию. Однако обнаружить вхождение подстроки `'ruby'` в строку `'Hello, world!'` не удается, и метод возвращает `nil`. В результате возникает ошибка: объект `nil` не поддерживает метод `odd?` для проверки нечетности числа.

В описанной ситуации, когда мы просто проверяем истинность выражения, нас вполне устроил в качестве результата `nil`, который в Ruby интерпретируется как `false`. В таких ситуациях вместо оператора точки используется оператор `&.`, который проигнорирует вызов метода и просто вернет `nil`:

```
 > hello = 'Hello, world!'
=> "Hello, world!"
 > hello.index('ruby')&.odd?
=> nil
```

Оператор безопасного вызова `&.` используется только в отношении `nil`, он не работает с другими объектами языка:

```
> hello&.odd?
NoMethodError (undefined method `odd?' for "Hello, world!":String)
> 3.&index('ruby')
NoMethodError (undefined method `index' for main:Object)
```

7.8. Ключевое слово *defined?*

Ключевое слово `defined?` позволяет проверить существование объекта в Ruby-программе. Если объект определен, `defined?` возвращает для него текстовое описание, иначе возвращается неопределенное значение `nil`:

```
 > defined? hello
=> nil
 > hello = 'world'
=> "world"
 > defined? hello
=> "local-variable"
 > defined? 1 + 1
=> "method"
 > defined? Hash
=> "constant"
 > @hello = 1
=> 1
 > defined? @hello
=> "instance-variable"
 > defined? $stdout
=> "global-variable"
```

Помимо ключевого слова `defined?`, язык программирования Ruby предоставляет отдельные методы для проверки существования различных типов переменных и констант (табл. 7.10).

***Таблица 7.10.** Методы проверки существования переменных и констант*

Метод	Описание
`local_variable_defined?`	Проверяет существование локальной переменной `hello`
`instance_variable_defined?`	Проверяет существование инстанс-переменной `@hello`
`class_variable_defined?`	Проверяет существование переменной класса `@@hello`
`const_defined?`	Проверяет существование константы, например, `CONST`

В отличие от ключевого слова `defined?`, если переменная существует, специализированные методы возвращают `true`, иначе — `false`. Кроме того, в качестве аргумента они принимают не сами объекты, а их названия в виде строки или символа:

```
 > hello = 'world'
=> "world"
 > defined? hello
=> nil
 > binding.local_variable_defined? :hello
=> true
 > @hello = 1
=> 1
 > instance_variable_defined? :@hello
=> true
 > Module.const_defined? :Hash
=> true
 > @@hello = 1
=> 1
 > Module.class_variable_defined? :@@hello
=> true
 > Module.class_variable_defined? '@@hello'
=> true
 > Module.class_variable_defined? '@@none'
=> false
```

7.9. Приоритет операторов

В заключение главы приведем таблицу приоритетов выполнения операторов (табл. 7.11). Операторы с более низким приоритетом расположены в таблице ниже, с более высоким — выше. Операторы, расположенные на одной строке, имеют одинаковый приоритет, и выполняется в первую очередь тот из операторов, который встречается в выражении первым.

Замечание

В этой главе рассмотрены не все операторы, представленные в табл. 7.11. Остальные операторы будут рассмотрены в последующих главах.

Таблица 7.11. *Приоритет операторов*

Оператор	Описание
`! ~ +`	Логическое отрицание `!`, поразрядное отрицание `~`, унарный плюс `+` (например, `+10`)
`**`	Возведение в степень
`-`	Унарный минус `-` (например, `-10`)
`* / %`	Умножение `*`, деление `/`, взятие остатка `%`
`+ -`	Арифметические плюс `+` и минус `-`
`>> <<`	Поразрядные сдвиги вправо `>>` и влево `<<`
`&`	Поразрядное И
`^ |`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ, поразрядное ИЛИ
`<= < > >=`	Операторы сравнения с более высоким приоритетом
`<=> == === != =~ !~`	Операторы сравнения с более низким приоритетом
`&&`	Логическое И
`||`	Логическое ИЛИ
`.. ...`	Операторы диапазона
`z ? y : z`	Тернарный оператор
`= %= { /= -= += |= &= >>= <<= *= &&= ||= **=`	Операторы присваивания
`not`	Логическое отрицание
`or and`	Логические ИЛИ и И

Задания

1. Создайте скрипт, который бы возводил одно целое число в другое, не прибегая к оператору `**` или методу `pow`.
2. Создайте скрипт, который бы определял четность или нечетность числа только при помощи поразрядных операторов.
3. Есть две переменные: `fst = 10` и `snd = 20`. Поменяйте значение переменных местами. Постарайтесь использовать минимальное количество кода.
4. Используя оператор `%`, выведите число π (3.14159265358979) с точностью до второго знака после запятой (3.14).

5. При помощи поразрядных операторов закодируйте в целом числе параметры геометрических фигур:

   - форма: круг, треугольник, квадрат, точка;
   - координаты центра фигуры в двумерной системе координат (x, y) в диапазоне от 0 до 255;
   - цвет фигуры: красный, оранжевый, желтый, зеленый, голубой, синий и фиолетовый.

   Одно целое число должно представлять одну фигуру и все ее характеристики.

ГЛАВА 8

Ветвление

Файлы с исходными кодами этой главы находятся в каталоге *conditions* сопровождающего книгу электронного архива.

Ветвление является одной из базовых возможностей всех современных языков программирования. В зависимости от выполнения или неыполнения условий, ветвление позволяет выполнить тот или иной набор выражений, объединенных составным оператором.

Решение о том, какой из участков кода выполнять, а какой игнорировать, принимается зачастую при помощи логических значений и выражений. Логические значения могут принимать только два состояния: «истина» — `true` и «ложь» — `false`. Они были рассмотрены в *главе 4*. А эта глава посвящена конструкциям ветвления: `if`, `unless`, `case` и тернарному оператору `x ? : y : z`. Кроме того, мы рассмотрим здесь логические операторы.

8.1. Ключевое слово *if*

Конструкция `if` позволяет проигнорировать или выполнить выражения в зависимости от аргумента (`true` или `false`), который следует сразу за ключевым словом `if`. Для обозначения конца конструкции используется ключевое слово `end` (листинг 8.1).

Листинг 8.1. Конструкция `if`. Файл if.rb

```
puts 'Начало программы'

if true
  puts 'Содержимое if-конструкции'
end

puts 'Завершение программы'
```

Между ключевыми словами `if` и `end` могут располагаться одно или несколько Ruby-выражений, которые составляют *тело* `if`-конструкции. Результатом выполнения программы из листинга 8.1 будут следующие строки:

```
Начало программы
Содержимое if-конструкции
Завершение программы
```

Если конструкции `if` в качестве аргумента передать значение `false`, то интерпретатор проигнорирует содержимое `if`-блока, и выполнение программы продолжится после ключевого слова `end` (листинг 8.2).

Листинг 8.2. Содержимое `if`-блока не будет выполнено. Файл if_false.rb

```
puts 'Начало программы'

if false
  puts 'Содержимое if-конструкции'
end

puts 'Завершение программы'
```

Результатом выполнения программы из листинга 8.2 будут следующие строки:

```
Начало программы
Завершение программы
```

На практике конструкция `if`, как правило, принимает не сами объекты `true` и `false`, а логические выражения. Например, при помощи константы `RUBY_VERSION` можно получить версию Ruby. Если сравнить значение константы с аналогичной строкой, можно получить `true`, если значение будет отличаться, мы получим `false`:

```
 > RUBY_VERSION
=> "2.5.3"
 > RUBY_VERSION == '2.5.3'
=> true
 > RUBY_VERSION == '2.6.0'
=> false
```

Выражение после ключевого слова `if` называется *условием*. В листинге 8.3 в качестве условия используется подобранное ранее выражение с участием константы `RUBY_VERSION`. Для того чтобы содержимое тела `if`-конструкции сработало, вам потребуется модифицировать код программы, задав корректную версию Ruby, установленную на вашем компьютере.

Листинг 8.3. Использование логического условия. Файл if_condition.rb

```
puts 'Начало программы'

if RUBY_VERSION == '2.5.3'
  puts 'Содержимое if-конструкции'
end

puts 'Завершение программы'
```

В качестве логического значения может выступать любое Ruby-выражение. Результат, который возвращается выражением, будет неявно приведен к логическому значению. Объекты `false` и `nil` рассматриваются как «ложь», все остальное будет рассматриваться как «истина» (листинг 8.4).

Листинг 8.4. Неявное приведение Ruby-выражений в `if`. Файл if_object.rb

```
if Object
  puts 'Класс неявно преобразуется к true'
end

if Object.new
  puts 'Объект так же неявно преобразуется к true'
end

if nil
  puts 'А вот nil — это false'
end

if puts 'Эта строка выведется'
  puts 'А эта уже нет'
end
```

Результатом работы программы из листинга 8.4 будут следующие строки:

```
Класс неявно преобразуется к true
Объект так же неявно преобразуется к true
Эта строка выведется
```

Первые две `if`-конструкции выполняют выражения в теле, т. к. класс `Object` и объект `Object.new` рассматриваются в контексте `if` как «истина», две последние `if`-конструкции игнорируют свои тела, т. к. `nil`, в том числе возвращаемый методом `puts`, рассматривается как «ложь».

8.1.1. Ключевые слова *else* и *elsif*

Иногда необходимо, чтобы в зависимости от условия, переданного конструкции `if`, выполнялся разный код: для «истины» `true` — один, для «лжи» `false` — другой. На такой случай предусмотрено другое ключевое слово: `else`. В листинге 8.5, если на компьютере установлена версия Ruby 2.5.3, выводится фраза: `'Корректная версия Ruby'`, в противном случае, если условие в конструкции `if` возвращает `false`, выводится фраза: `'Некорректная версия Ruby'`.

Листинг 8.5. Использование ключевого слова `else`. Файл if_else.rb

```
if RUBY_VERSION == '2.5.3'
  puts 'Корректная версия Ruby'
```

```
else
  puts 'Некорректная версия Ruby'
end
```

Иногда недостаточно двух состояний, которые обеспечивают ключевые слова `if` и `else`. Для добавления в `if`-конструкцию дополнительных условий предназначено ключевое слово `elsif` (листинг 8.6).

Листинг 8.6. Использование ключевого слова `elsif`. Файл if_elsif.rb

```
if RUBY_VERSION >= '2.5.3'
  puts 'Корректная версия Ruby'
elsif RUBY_VERSION >= '2.4.0'
  puts 'Проблемная версия Ruby'
else
  puts 'Некорректная версия Ruby'
end
```

В программе из листинга 8.6 сначала вычисляется первое условие: `RUBY_VERSION >= '2.5.3'`. Если оно истинно и возвращает `true`, выводится фраза `'Корректная версия Ruby'`, после чего интерпретатор Ruby перемещается к ключевому слову `end`. В случае, если условие ложно и возвращает `false`, проводится проверка условия в ключевом слове `elsif`. Если условие: `RUBY_VERSION >= '2.4.0'` оказывается истинным, выводится фраза `'Проблемная версия Ruby'` и интерпретатор перемещается к ключевому слову `end`. Если это дополнительное условие также оказывается ложным, выполняется код из `else`-блока (рис. 8.1).

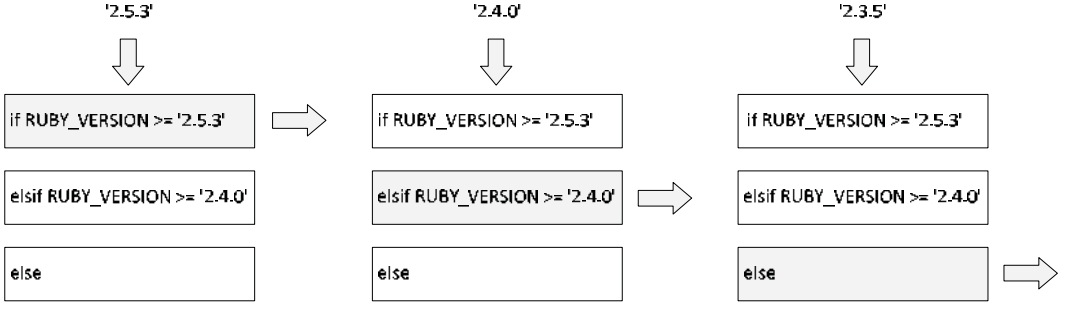


Рис. 8.1. Логика работы ключевых слов `else` и `elsif`

В конструкции `if` ключевые слова `else` и `elsif` являются необязательными. Количество ключевых слов `elsif` не ограничено — можно вводить любое количество новых условий. Ключевое слово `else` допускается лишь в одном экземпляре.

8.1.2. Ключевое слово *then*

У оператора `if` имеется ключевое необязательное слово `then`, которое можно указать после условия (листинг 8.7).

Листинг 8.7. Использование ключевого слова `then`. Файл if_then.rb

```
if RUBY_VERSION == '2.5.3' then
  puts 'Содержимое if-конструкции'
end
```

На практике `then` практически всегда опускается. Ruby-разработчики придерживаются минималистского стиля: если можно что-то опустить, этот элемент синтаксиса всегда опускается, среди альтернатив всегда выбирается самый короткий вариант.

Исключение составляет ситуация, когда `if`-условие необходимо вытянуть в одну строку (листинг 8.8).

Листинг 8.8. Однострочный режим конструкции `if`. Файл if_then_one_line.rb

```
if RUBY_VERSION == '2.5.3' then puts 'Содержимое if-конструкции' end
```

Хотя такой подход не рекомендуется в промышленном коде, при экспериментах в интерактивном Ruby часто практикуется однострочный режим разработки. Исправить ошибку в одной, пусть и длинной, строке проще, чем снова набрать исходный код программы.

Листинг 8.9. Ошибочное использование конструкции `if`. Файл if_then_error.rb

```
if RUBY_VERSION == '2.5.3' puts 'Содержимое if-конструкции' end
```

Если в однострочном режиме убрать ключевое слово `then` (листинг 8.9), интерпретатор Ruby не сможет отделить условие от тел, и произойдет ошибка:

```
$ ruby if_then_error.rb
if_then_error.rb:1: syntax error, unexpected tIDENTIFIER, expecting
keyword_then or ';' or '\n'
if RUBY_VERSION == '2.5.3' puts 'Содержимое if-к...
                           ^~~~
```

8.1.3. *if*-модификатор

Конструкция `then`, рассмотренная в предыдущем разделе, никогда не используется на практике. Если тело `if`-блока состоит из одного небольшого выражения, как правило используется `if`-модификатор. В этом случае ключевое слово `if` и условие помещаются в конце Ruby-выражения (листинг 8.10).

> ***Замечание***
>
> `if`-модификатор был заимствован из языка Perl, в котором он впервые появился. Такая форма конструкции `if` интенсивно используется на практике.

Листинг 8.10. `if`-модификатор. Файл if_modifier.rb

```
puts 'Содержимое if-конструкции' if RUBY_VERSION == '2.5.3'
```

8.1.4. Присваивание *if*-результата переменной

Конструкция `if` возвращает значение, в качестве которого выступает результат вычисления последнего выражения в блоке `if` или `else` (в зависимости от того, какой из них срабатывает). В листинге 8.11 результат, возвращаемый `if`-конструкцией, передается методу `puts` для вывода в стандартный поток вывода.

Листинг 8.11. Конструкция `if` возвращает значение. Файл if_return.rb

```
puts (if RUBY_VERSION == '2.5.3'
        'Корректная версия'
      else
        'Некорректная версия'
      end)
```

Иногда внутри `if`-конструкции одной и той же переменной присваиваются значения в зависимости от условия (листинг 8.12).

Листинг 8.12. Инициализация переменной внутри `if`. Файл if_var_incorrect.rb

```
if RUBY_VERSION == '2.5.3'
  var = 'Корректная версия'
else
  var = 'Некорректная версия'
end

puts var
```

Код в листинге 8.12 хотя и синтаксически верен, обычно заменяется присваиванием переменной результата вычисления конструкции `if` (листинг 8.13).

Листинг 8.13. Корректная инициализация переменной. Файл if_var_correct.rb

```
var = if RUBY_VERSION == '2.5.3'
        'Корректная версия'
      else
        'Некорректная версия'
      end

puts var
```

При таком подходе переменная упоминается в программе лишь один раз, что позволяет сократить вероятность возникновения ошибок, — когда, например, при переименовании переменной название переменной исправляется в одном месте, а во втором месте по ошибке остается старым.

8.1.5. Присваивание в условии *if*-оператора

Переменным можно присваивать значение не только внутри тела `if`-конструкции или как результат вычисления тела. Можно присваивать значение переменной непосредственно в условии (листинг 8.14).

Листинг 8.14. Корректная инициализация переменной. Файл if_var_condition.rb

```
if x = 1
  y = 'hello'
end
puts x # 1
puts y # hello
```

В листинге 8.14 в условии оператора `if` переменной `x` присваивается значение `1`. Поскольку оператор возвращает присвоенное значение, и оно отлично от `false` и `nil`, в контексте `if`-условия значение `1` рассматривается как «истина» `true`. В результате выражение внутри тела `if`-конструкции срабатывает, и переменной `y` присваивается значение строки `'hello'`.

Если запустить программу на выполнение, в качестве результата можно получить следующие строки:

```
$ ruby if_var_condition.rb
if_var_condition.rb:1: warning: found = in conditional, should be ==
1
hello
```

Помимо ожидаемых строк со значениями переменных `x` и `y`, интерпретатор Ruby выводит предупреждение, сообщающее о том, что в условии `if` используется оператор равенства `=`, а, возможно, имелся в виду оператор сравнения. Интерпретатор «подозревает», что переменная `x` сравнивалась со значением `1`, и предупреждает о возможной опечатке.

Несмотря на возникающее предупреждение, такой прием весьма часто используется на практике. Например, для получения какого-либо ресурса — скажем, соединения с базой данных:

```
if conn = get_database_connection
  conn.sql(...)
  ...
end
```

Если метод `get_database_connection` вернул значение, отличное от `nil`, срабатывает тело `if`-конструкции, в котором можно выполнить запросы, будучи уверенными в корректном определении переменной `conn`. Если соединение не получено, тело `if`-конструкции будет проигнорировано.

Тем не менее, такой код будет генерировать предупреждения как от интерпретатора Ruby, так и от линтеров — например, того же `rubocop` (см. *главу 3*). Линтеру `rubocop`

можно сообщить, что оператор равенства используется намеренно, если заключить присваивание в круглые скобки:

```
if (conn = get_database_connection)
  conn.sql(...)
  ...
end
```

Даже если содержимое блока `if` не может быть достигнуто, интерпретатор заводит такую переменную, назначая ей значение `nil` (листинг 8.15).

Листинг 8.15. Инициализация переменной внутри `if`-блока. Файл if_assign.rb

```
if false
  y = 'Hello, world!'
end
p y # nil
p z # undefined local variable or method `z' for main:Object (NameError)
```

Как видно из листинга 8.15, переменная `y` получает значение `nil`, несмотря на то, что содержимое `if`-блока было проигнорировано. В то же время попытка обратиться к переменной `z` завершается сообщением об ошибке, что такая переменная не существует.

Дело в том, что на самом деле интерпретатор перед выполнением программы предварительно несколько раз ее сканирует. Во время одного из таких прогонов создаются все переменные, для которых была обнаружена инициализация при помощи оператора присваивания =. Поэтому при выполнении программы переменная `y` заведена, но не инициализирована.

8.2. Логические операторы

Несколько простых условий допускается объединять при помощи логических операторов в сложные логические конструкции (табл. 8.1). Полученные логические выражения потом можно использовать совместно с конструкцией `if`.

***Таблица 8.1.** Логические операторы*

Оператор	Описание
`x && y`	Логическое И возвращает `true`, если оба операнда: `x` и `y` — истинны, в противном случае возвращается `false`
`x and y`	Логическое И, отличающееся от оператора `&&` меньшим приоритетом
`x || y`	Логическое ИЛИ возвращает `true`, если хотя бы один из операндов: `x` и `y` — истинен. Если оба операнда ложны, оператор возвращает `false`
`x or y`	Логическое ИЛИ, отличающееся от оператора `||` меньшим приоритетом
`! x`	Возвращает либо `true`, если `x` ложен, либо `false`, если `x` истинен

Логические операторы очень похожи на поразрядные операторы (см. *главу 7*), только вместо отдельных битов числа они оперируют логическими значениями.

8.2.1. Логическое И. Оператор &&

Логическое И записывается как `&&`. В табл. 8.2 в левой колонке представлен левый операнд, в верхней строке — правый, на пересечении приводится результат.

***Таблица 8.2.** Правила преобразования операндов оператором &&*

Левый операнд	Правый операнд	
	true	**false**
true	`true`	`false`
false	`false`	`false`

Оператор возвращает `true`, если оба операнда равны `true`, и `false` в любом другом случае. Проще всего продемонстрировать использование оператора `&&` в интерактивном Ruby:

```
 > true && true
=> true
 > true && false
=> false
 > false && true
=> false
 > false && false
=> false
```

Пример использования оператора `&&` приведен в листинге 8.16. При помощи метода `rand` два раза подряд извлекается случайное число от `0` до `1`, которое сохраняется в две переменные: `fst` и `snd`. Далее в `if`-условии проверяется каждое из чисел (сравнивается с единицей). Условия объединяются при помощи оператора `&&`.

Листинг 8.16. Использование оператора && . Файл and.rb

```
fst = rand(0..1)
snd = rand(0..1)
if fst == 1 && snd == 1
  puts "Оба значения равны единице: #{fst} и #{snd}"
else
  puts "Пока не получилось: #{fst} и #{snd}"
end
```

8.2.2. Логическое ИЛИ. Оператор ||

Помимо логического И, Ruby предоставляет логическое ИЛИ, которое записывается как две вертикальные черты `||` (табл. 8.3).

Таблица 8.3. *Правила преобразования операндов оператором* **||**

Левый операнд	Правый операнд	
	true	**false**
true	`true`	`true`
false	`true`	`false`

Оператор возвращает `false` только в том случае, если оба операнда равны `false`, во всех остальных случаях возвращается `true`.

Оператор `||` является «ленивым» — если первый операнд равен `true`, то в вычислении второго операнда отсутствует надобность: независимо от того, будет он равен `true` или `false`, логическое выражение все равно примет значение `true`.

В логических операторах любой объект интерпретируется как логическое значение, при этом сам оператор не приводит результат к `true` или `false`:

```
 > true || 'world'
=> true
 > false || 'world'
=> "world"
```

Эти правила открывают широкие возможности для сокращения кода и выбора значения по умолчанию (листинг 8.17).

Листинг 8.17. Использование оператора `||`. Файл or.rb

```
item = 'Ruby'
puts "Hello, #{item || 'world'}!" # Hello, Ruby!

item = nil
puts "Hello, #{item || 'world'}!" # Hello, world!
```

В листинге 8.17 вообще не используются объекты `true` и `false`. Вместо `false` выступает объект `nil`, вместо `true` — все остальные объекты Ruby. Если переменная `item` принимает неопределенное значение `nil`, выражение `item || 'world'` возвращает строку `'world'`. Если переменной `item` назначен объект, отличный от `false` и `nil`, выражение `item || 'world'` вернет `item`.

Для логических И и ИЛИ допускается сокращенная форма операторов присваивания: `&&=` и `||=` (рис. 8.2).

Последняя форма (`||=`) очень распространена среди Ruby-программистов, т. к. позволяет присвоить неопределенной переменной значение по умолчанию (листинг 8.18).

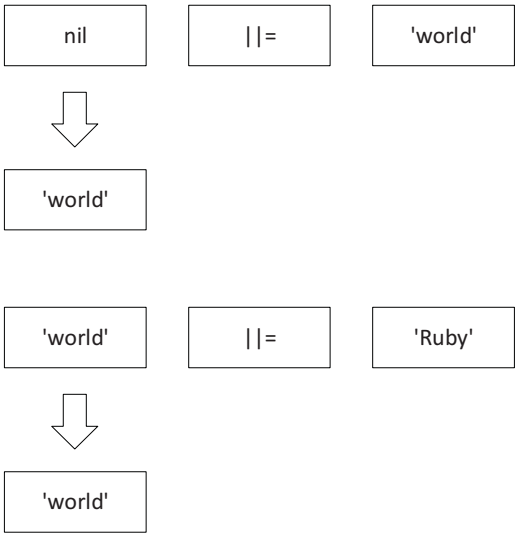


Рис. 8.2. Логика работы идиомы `||=`

Листинг 8.18. Использование оператора `||=`. Файл or_equal.rb

```
item = nil

item ||= 'world'
puts "Hello, #{item}!" # Hello, world!

item ||= 'Ruby'
puts "Hello, #{item}!" # Hello, world!
```

В листинге 8.18 оператор `||=` встречается два раза. При его первом использовании, когда переменная `item` имеет неопределенное значение `nil`, оператор назначает переменной значение `'world'`. Однако повторный вызов `||=` не приводит к изменению состояния переменной `item`. Запишем оператор в развернутой форме:

```
item = item || 'world'
```

Если переменная `item` принимает значение `nil` или `false`, интерпретатор вынужден вычислять второй операнд, чтобы определить конечное значение. Выражение `item || 'world'` возвращает `'world'`. При этом переменная `item` имеет уже какое-то значение, которое рассматривается как «истина», и нет надобности в вычислении второго операнда — каким бы он ни оказался, выражение все равно останется истинным. В этом случае происходит присваивание значения переменной самой себе: `item = item`.

Если результат, который необходимо присвоить переменной, получается в результате длительного вычисления, требующего нескольких выражений, можно воспользоваться ключевыми словами `begin` и `end`. Выражения, заключенные между этими ключевыми словами, рассматриваются как единое целое (листинг 8.19).

Листинг 8.19. Использование ключевых слов `begin` и `end`. Файл begin_end.rb

```
item ||= begin
  x = 12 - 5
  y = 15 - 2
  sum = x * x + y * y
  sum ** 0.5
end

puts item
```

Переменная получает в качестве значения результат, который возвращает последнее выражение в блоке `begin ... end`.

Операторы `&&` и `||` имеют альтернативу в виде операторов `and` и `or`, обладающих более низким приоритетом. То есть, если в арифметических операциях сначала выполняется умножение и деление и лишь затем сложение и вычитание, то в логических операторах сначала выполняются `&&` и `||` и лишь затем `and` и `or`. Однако, в отличие от языка Perl, широкого распространения эти операторы на практике не получили.

8.2.3. Логическое отрицание

Иногда в логических выражениях необходимо задействовать отрицание. Для этого можно использовать оператор неравенства (листинг 8.20).

Листинг 8.20. Использование оператора неравенства. Файл not_equal.rb

```
if RUBY_VERSION != '2.4.0'
  puts 'Не корректная версия'
end
```

Это не единственный способ логического отрицания — вместо оператора `!=` можно использовать ключевое слово `not`:

```
 > not true
=> false
 > not false
=> true
 > not Object
=> false
```

В листинге 8.21 приводится пример программы, в котором оператор неравенства `!=` заменен на комбинацию ключевого слова `not` и оператора равенства `==`.

Листинг 8.21. Использование ключевого слова `not`. Файл not.rb

```
if not RUBY_VERSION == '2.4.0'
  puts 'Не корректная версия'
end
```

Вместо ключевого слова `not` чаще применяется оператор восклицательного знака `!`, который отличается от `not` более высоким приоритетом. Поэтому совместно с ним, как правило, используются круглые скобки (листинг 8.22).

Листинг 8.22. Использование оператора `!`. Файл not_opertator.rb

```
if !(RUBY_VERSION == '2.4.0')
  puts 'Некорректная версия'
end
```

Без круглых скобок оператор `!` в силу своего высокого приоритета будет применен к константе `RUBY_VERSION`, в результате чего мы получим объект `false`, который затем будет сравниваться со строкой `'2.4.0'`:

```
 > !RUBY_VERSION == '2.4.0'
=> false
 > false == '2.4.0'
=> false
```

8.3. Ключевое слово *unless*

Вместо оператора неравенства `!=` или логического отрицания `!` можно использовать ключевое слово `unless`, которое является полной противоположностью `if`.

В листинге 8.23 производятся два вызова `if`: с отрицанием и с `unless` — логически они полностью эквивалентны.

Листинг 8.23. Использование ключевого слова `unless`. Файл unless.rb

```
unless RUBY_VERSION == '2.4.0'
  puts 'Некорректная версия'
end

if !(RUBY_VERSION == '2.4.0')
  puts 'Некорректная версия'
end
```

Ключевое слово `unless` поддерживает все возможности и модификации `if` (листинг 8.24).

Листинг 8.24. Использование ключевого слова `unless`. Файл unless_modifier.rb

```
puts 'Некорректная версия' unless RUBY_VERSION == '2.4.0'
```

8.4. Условный оператор

Для сокращения конструкции `if ... else ... end` часто используется *условный* или *тернарный* оператор `X ? Y : Z`. Если выражение `X` истинное, оператор возвращает `Y`, если ложное — `Z` (рис. 8.3).

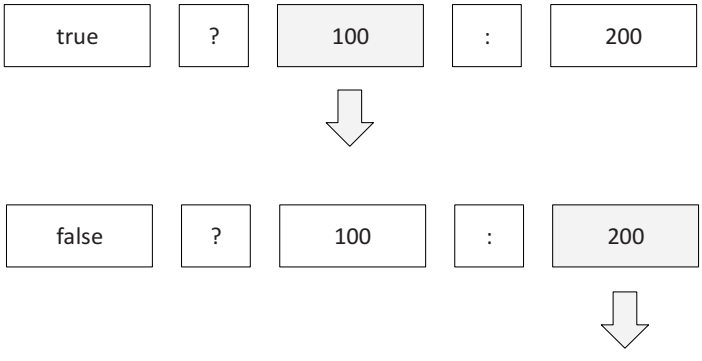


Рис. 8.3. Логика работы тернарного оператора

Операции присваивания в листинге 8.25 полностью одинаковые, однако условный оператор занимает меньший объем.

Листинг 8.25. Использование условного оператора. Файл ternary.rb

```
var = if rand(0..1) == 0
        100
      else
        200
      end

var = rand(0..1) == 0 ? 100 : 200
```

Основное назначение условного оператора — сократить конструкцию `if` до одной строки, если это не приводит к снижению читабельности.

8.5. Ключевое слово *case*

Ключевое слово `case` так же позволяет переключать логику программы — оно похоже на `if` с большим количеством `elsif`-блоков (листинг 8.26).

> ***Замечание***
>
> Во многих C-подобных языках имеется конструкция `switch`, которая очень походит на конструкцию `case` из Ruby. Тем не менее в синтаксисе и работе `case` имеются отличия от `switch`, на которые следует обратить внимание, если вы уже знакомы с одним или несколькими C-подобными языками.

Листинг 8.26. Использование конструкции case. Файл case.rb

```
number = rand(0..2)

case number
when 0
  puts 'Ноль';
when 1
  puts 'Единица'
else
  puts 'Два'
end
```

В листинге 8.26 представлена программа, которая генерирует случайное число: `0`, `1` или `2`. Полученное значение помещается в переменную `number`, которая, в свою очередь, передается в качестве аргумента ключевому слову `case`.

Между ключевым словом `case` и `end` располагаются `when`-условия. Ключевым словам `when` передаются аргументы, с которыми последовательно сравнивается `number`. Если находится совпадение, выполняются выражения `when`-блока.

Если ни одно из `when`-условий не подошло, срабатывает `else`-блок, который, как и в случае `if` и `unless`, является необязательным.

Ключевое слово `when` допускает в качестве аргумента любой объект языка Ruby. В листинге 8.27 в качестве аргументов используются диапазоны.

Листинг 8.27. Использование диапазонов в конструкции case. Файл case_range.rb

```
number = rand(0..100)

case number
when 0..50
  puts 'От нуля до 50';
when 51..100
  puts 'От 51 до 100'
else
  puts 'Число не входит в диапазон от 0 до 100'
end
```

Программа из листинга 8.27 получает случайное значение в диапазоне от `0` до `100` и при помощи конструкции `case` определяет, больше или меньше это число, чем `50`.

Эту программу можно попробовать переписать при помощи конструкции `if` (листинг 8.28).

Листинг 8.28. Файл case_elsif_incorrect.rb

```
number = rand(0..100)

if (0..50) == number
  puts 'От нуля до 50';
```

```
elsif (51..100) == number
  puts 'От 51 до 100'
else
  puts 'Число не входит в диапазон от 0 до 100'
end
```

Однако в этом случае программа «сломается» — при каждом запуске будет выводиться фраза `'Число не входит в диапазон от 0 до 100'`. Связано это с тем, что сравнение при помощи оператора == двух разных объектов возвращает `false` (см. *разд. 7.5.1*). Для того чтобы конструкция `if` заработала точно так же, как `case`, потребуется заменить оператор == на === (листинг 8.29).

Листинг 8.29. Файл case_elsif_correct.rb

```
number = rand(0..100)

if (0..50) === number
  puts 'От нуля до 50';
elsif (51..100) === number
  puts 'От 51 до 100'
else
  puts 'Число не входит в диапазон от 0 до 100'
end
```

То есть при сравнении в конструкции `case` используется оператор ===. За счет того, что оператор === перегружен во многих объектах (например, классах), это позволяет строить весьма элегантные конструкции — например, сравнивать объект с классом, определяя, к какому классу он принадлежит (листинг 8.30).

Листинг 8.30. Определение класса объекта. Файл case_class.rb

```
number = 'Hello, world!'

type = case number
       when Integer
         'Целое число'
       when String
         'Строка'
       when Range
         'Диапазон'
       else
         'Какой-то класс'
       end

puts type # Строка
```

Результат вычисления `case`, как и в случае `if` или `unless`, можно присвоить в качестве значения переменной (в примере это переменная `type`).

Если конструкция when содержит лишь одно выражение, ее можно вытянуть в одну строку за счет использования ключевого слова then (листинг 8.31).

Листинг 8.31. Использование ключевого слова then. Файл case_when_then.rb

```
number = 1..100

type = case number
       when Integer then 'Целое число'
       when String then 'Строка'
       when Range then 'Диапазон'
       else 'Какой-то класс'
       end

puts type # Диапазон
```

8.6. Советы

В программах следует избегать избыточного вложения. В листинге 8.32 два if-блока вкладываются друг в друга. Это позволяет избежать использования логических операторов для соединения условий, однако приводит к дублированию содержимого else-блоков.

Листинг 8.32. Избыточное вложение. Файл nested_wrong.rb

```
flag1 = true
flag2 = true

if flag1
  if flag2
    puts 'Оба флага истинны'
  else
    puts 'Условие: false (Один из флагов ложен)'
  end
else
  puts 'Условие: false (Один из флагов ложен)'
end
```

Если в этом случае воспользоваться логическим объединением, можно значительно сократить объем программы и избавиться от повторов в else-блоках (листинг 8.33).

Листинг 8.33. Сокращение кода за счет использования &&. Файл nested_correct.rb

```
flag1 = true
flag2 = true
```

```
if flag1 && flag2
  puts 'Оба флага истинны'
else
  puts 'Условие: false (Один из флагов ложен)'
end
```

Еще одной распространенной ошибкой является использование прямого сравнения с объектами `true` и `false` (листинг 8.34).

Листинг 8.34. Прямое сравнение с `true` — это ошибка. Файл true_compare.rb

```
var = nil

if var.nil? == true
  puts 'Переменная не инициализирована'
end
```

Такой подход порождает уродливые конструкции вроде `!!`, когда при помощи двойного отрицания значение, которое в контексте ключевого слова `if` и так воспринимается как логическое, мы вынуждены приводить явно к `true` или к `false` (листинг 8.35).

Листинг 8.35. Не следует использовать двойное отрицание. Файл true_cast.rb

```
if !!Object == true
  puts 'Object - это истина'
end
```

Всего этого можно избежать, если пользоваться соглашениями языка Ruby: `nil` и `false` — рассматривается как «ложь», все остальное — «истина». В этом случае приведенные ранее некорректные примеры можно переписать более элегантно (листинг 8.36).

Листинг 8.36. Использование соглашений. Файл true.rb

```
var = nil

if var.nil?
  puts 'Переменная не инициализирована'
end

if Object
  puts 'Object - это истина'
end
```

Задания

1. Создайте метод `colors`, который предоставляет справочник цветов радуги (1 — красный, 2 — оранжевый, 3 — желтый, 4 — зеленый, 5 — голубой, 6 — синий, 7 — фиолетовый). Метод должен принимать номер и возвращать цвет, соответствующий этому номеру. В случае ошибочного номера должно возвращаться значение `nil`.
2. Создайте метод `week`, который предоставляет справочник дней недели (1 — понедельник, 2 — вторник, 3 — среда, 4 — четверг, 5 — пятница, 6 — суббота, 7 — воскресенье). Метод должен принимать номер и возвращать название для недели, соответствующее этому номеру. В случае ошибочного номера должно возвращаться значение `nil`.
3. Создайте программу number.rb, которая принимает в качестве аргумента три целых числа — например: `number.rb 15 7 20`. Проверьте, что переданные числа действительно целые. В качестве результата программа должна возвращать максимальное значение среди чисел.
4. Создайте программу check.rb, которая принимает единственный аргумент. Определите, что пользователь ввел в качестве аргумента: строку, целое или вещественное число?
5. Создайте программу posneg.rb, которая принимает в качестве аргумента число и сообщает, является оно положительным или отрицательным. В случае, если первый аргумент программы числом не является, программа должна вывести сообщение «Это не число».
6. Создайте программу evenodd.rb, которая принимает в качестве аргумента число и сообщает, является оно четным или нечетным. В случае, если первый аргумент программы числом не является, программа должна вывести сообщение «Это не число».

ГЛАВА 9

Глобальные методы

Файлы с исходными кодами этой главы находятся в каталоге *methods* сопровождающего книгу электронного архива.

В Ruby почти все является либо объектом, либо методом объекта. *Метод* предназначен для группировки повторяющихся действий, которые затем можно вызывать путем обращения к имени метода.

Методы — весьма обширная тема, рассмотреть все аспекты их работы в одной главе не получится. Поэтому текущая глава будет посвящена *глобальным методам*. Мы подробно рассмотрим приемы работы как с готовыми методами, так и разработку своих собственных методов.

9.1. Создание метода

Для того чтобы создать метод, необходимо объявить его при помощи ключевого слова `def`, после которого указывается название метода (в snake-режиме). Завершается метод ключевым словом `end`. Все, что расположено между ключевыми словами `def` и `end`, является *телом* метода. В теле метода допускается размещение любых Ruby-выражений.

В листинге 9.1 создается метод, который конвертирует 5 килограммов в граммы.

Листинг 9.1. Создание метода. Файл method.rb

```
def convert
  5 * 1000
end

puts convert # 5000
```

Для того чтобы выполнить метод, его необходимо вызывать, обратившись к нему по его имени. Метод возвращает значение — чтобы его увидеть, необходимо вывести его в стандартный поток вывода при помощи `puts`.

9.2. Параметры и аргументы

Если вместо пяти потребуется сконвертировать в граммы 11 килограммов, понадобится либо исправить содержимое метода `convert`, либо написать другой метод, который будет конвертировать в граммы уже 11 килограммов. Это не очень удобно и приводит к тому, что код начинает повторяться. Для решения этой проблемы предусмотрена *параметризация методов*. Параметры передаются в круглых скобках после имени метода. В листинге 9.2 методу передается параметр `value`, который используется вместо значения `5`. Параметры выступают в методе, как обычные локальные переменные.

> ***Замечание***
>
> Следует обратить внимание на терминологию — локальные переменные, которые указываются в круглых скобках после названия метода, называются *параметрами*. Значения, которые указываются в круглых скобках при вызове метода, называются *аргументами*.

Листинг 9.2. Использование параметра. Файл param.rb

```
def convert(value)
  value * 1000
end

puts convert(11) # 11000
```

При вызове метода `convert` ему передается один аргумент — число `11`. Это число становится значением параметра `value`, и выражение внутри метода возвращает уже новое значение `11000`.

Если в листинге 9.2 не указать аргумент метода, возникнет сообщение об ошибке:

```
> convert
param.rb:1:in `convert': wrong number of arguments (given 0, expected 1)
                                                              (ArgumentError)
```

Оно информирует, что при вызове метода `convert` ожидается один аргумент, в то время как передано `0` аргументов.

Методы могут принимать несколько параметров, которые перечисляются в круглых скобках через запятую. В методе `convert` можно заменить значение `1000` параметром `factor`, что позволит конвертировать не только килограммы в граммы, а, например, килобайты в байты (листинг 9.3).

Листинг 9.3. Использование нескольких параметров. Файл params.rb

```
def convert(value, factor)
  value * factor
end

puts convert(11, 1024) # 11264
```

9.2.1. Значения по умолчанию

Параметрам можно назначать значения по умолчанию. В этом случае, если аргумент не задан, параметру будет присвоено значение по умолчанию. В листинг 9.4 параметр `factor` имеет значение по умолчанию `1000`.

> ***Замечание***
>
> Круглые скобки можно опускать как при определении метода, так и при его вызове. Однако по соглашениям, принятым в Ruby-сообществе, при определении методов круглые скобки всегда указываются. При вызове метода выбор варианта со скобками или без них остается за разработчиком.

Листинг 9.4. Параметр по умолчанию. Файл default_param.rb

```
def convert(value, factor = 1000)
  value * factor
end

puts convert(11)       # 11000
puts convert(11, 1024) # 11264
```

При вызове метода можно не указывать аргумент, которому соответствует параметр со значением по умолчанию. Если аргумент указывается явно, он перезаписывает значение по умолчанию.

9.2.2. Неограниченное количество параметров

Некоторые методы допускают передачу им произвольного количества аргументов. Ярким представителем таких методов является `puts` (листинг 9.5).

Листинг 9.5. `puts` принимает произвольное количество аргументов. Файл puts.rb

```
puts 'hello', 'ruby', 'worlds'
```

Результатом вызова программы из листинга 9.5 будет вывод всех трех аргументов на отдельных строках:

```
hello
ruby
worlds
```

Мы можем самостоятельно разрабатывать такие методы. Для того чтобы метод мог принимать неограниченное количество параметров, необходимо воспользоваться оператором `*`, который размещается перед одним из параметров (листинг 9.6).

Листинг 9.6. Метод принимает произвольное количество параметров. Файл splat.rb

```
def multi_params(*params)
  p params # [1, 2, 3, 4]
end

multi_params(1, 2, 3, 4)
```

В листинге 9.6 параметр `params` передается методу `p`. Выполнив программу, можно убедиться, что `params` представляет собой массив. Наряду с параметрами, которые предваряет оператор `*`, можно использовать обычные параметры (листинг 9.7).

Листинг 9.7. Файл multi_params.rb

```
def multi_params(x, y, *params)
  p x      # 1
  p y      # 2
  p params # [3, 4]
end

multi_params(1, 2, 3, 4)
```

Параметр `x` принимает значение `1`, параметр `y` — `2`, параметру `params` достается урезанный массив из двух последних аргументов: `3` и `4`.

Аргументы метода можно упаковать в массив, разложив их в последовательность при помощи оператора `*` (листинг 9.8).

Листинг 9.8. Развертывание массива в список аргументов. Файл array_params.rb

```
def array_params(x, y, z)
  p x # 6
  p y # 3
  p z # 2
end

point = [6, 3, 2]
array_params(*point)
```

Метод `array_params` принимает три параметра, однако при вызове ему передается единственный массив `point`, элементы которого разворачиваются в список при помощи оператора `*`.

При попытке передать методу массив с большим или меньшим количеством элементов возникает ошибка (листинг 9.9).

Листинг 9.9. Неверное количество элементов в массиве. Файл array_params_nil.rb

```
def array_params(x, y, z)
  p x # 6
```

```
  p y # 3
  p z # 2
end

point = [6, 3, 2, 6]
array_params(*point)
# `array_params': wrong number of arguments (given 4, expected 3)

point = [6, 3]
array_params(*point)
# `array_params': wrong number of arguments (given 2, expected 3)
```

9.2.3. Позиционные параметры

Помимо обычных параметров можно использовать *позиционные*. Для этого в конце каждого параметра добавляется двоеточие (листинг 9.10).

Листинг 9.10. Позиционные параметры. Файл position.rb

```
def convert(value:, factor: 1000)
  value * factor
end

puts convert(value: 11)               # 11000
puts convert(value: 11, factor: 1024) # 11264
```

При вызове метода необходимо указывать название каждого параметра. Позиционные параметры дополнительно подсказывают, что означает тот или иной аргумент.

Кроме того, позиционные параметры позволяют менять порядок следования аргументов (листинг 9.11).

Листинг 9.11. Изменение порядка следования аргументов. Файл position_change.rb

```
def convert(value:, factor: 1000)
  value * factor
end

puts convert(value: 11, factor: 1024) # 11264
puts convert(factor: 1024, value: 11) # 11264
```

Так как позиционный аргумент представляет собой пару «ключ-значение», интерпретатор Ruby легко может сопоставить аргумент параметру.

9.2.4. Хэши в качестве параметров

Позиционные параметры очень похожи на Hash-объект с символами в качестве ключей. Использование хэшей тоже вполне допустимо в качестве параметра мето-

да. Более того, если хэш передается последним аргументом в методе, можно опускать фигурные скобки (листинг 9.12).

Листинг 9.12. Использование хэша в качестве аргумента. Файл hash.rb

```
def greeting(params)
  p params
end

greeting({first: 'world', second: 'Ruby'})
greeting(first: 'world', second: 'Ruby')
greeting first: 'world', second: 'Ruby'
```

Распространенным приемом при работе с хэшем в методе является удаление из него элементов (листинг 9.13). Для этого используется метод `delete`.

Листинг 9.13. Удаление элементов из хэша методом `delete`. Файл delete.rb

```
def greeting(params)
  puts params.delete :first  # world
  puts params.delete :second # Ruby
  p params                   # {:third=>"hello"}
end

greeting first: 'world', second: 'Ruby', third: 'hello'
```

9.3. Возвращаемое значение

Ruby поддерживает специальное ключевое слово `return` для возврата значений из методов (листинг 9.14).

Листинг 9.14. Использование ключевого слова `return`. Файл return.rb

```
def convert(value, factor)
  return value * factor
end

puts convert(11, 1024) # 11264
```

До этого момента ключевое слово `return` не использовалось, т. к. в нем не возникала необходимость. Дело в том, что метод и так возвращает результат, полученный в последнем выражении метода. Более того, соглашения, принятые в Ruby-сообществе, требуют исключать `return` в последней строчке метода.

Впрочем, ключевое слово `return` находит применение в случае, когда необходимо досрочно покинуть метод (листинг 9.15).

Листинг 9.15. Досрочный выход из метода. Файл before_return.rb

```
def convert(value, factor = nil)
  return value * 1000 unless factor
  value * factor
end

puts convert(11)       # 11000
puts convert(11, 1024) # 11264
```

В представленном варианте метода `convert` второй параметр `factor` принимает значение по умолчанию `nil`. В этом случае срабатывает модификатор `unless`, и при помощи ключевого слова `return` возвращается значение `value * 1000`. Оператор `return` осуществляет возврат из метода, и последнее выражение `value * factor` не выполняется.

Если второй параметр `factor` принимает значение, отличное от `nil` и `false`, оно рассматривается как «истина», и оператор `unless` игнорирует `return`-выражение. В этом случае метод возвращает результат вычисления последней строки: `value * factor`.

Метод может возвращать только один объект. В том случае, когда необходимо вернуть несколько значений, прибегают к коллекциям — например, к массивам. В листинге 9.16 метод `number` принимает в качестве единственного параметра `x` число и возвращает результат в виде массива с исходным значением `x`, его квадратом и значением квадратного корня.

Листинг 9.16. Возврат нескольких значений. Файл array_return.rb

```
def number(x)
  [x, x * x, x ** 0.5]
end

p number(7) # [7, 49, 2.6457513110645907]

arr = number(7)
puts arr[0] # 7
puts arr[1] # 49
puts arr[2] # 2.6457513110645907

original, square, sqrt = number(7)
puts original # 7
puts square   # 49
puts sqrt     # 2.6457513110645907
```

Результат работы метода может использоваться как массив, а может быть разложен на отдельные переменные с помощью операции параллельного присваивания (см. *разд. 7.3.2*).

9.4. Получатель метода

У метода всегда есть *получатель* — объект которому он адресован (рис. 9.1). Даже когда метод определяется в глобальной области видимости, метод адресуется объекту по умолчанию.

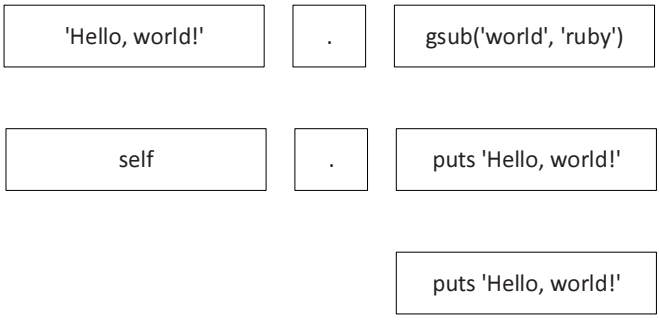


Рис. 9.1. У метода всегда имеется получатель, даже если он явно не указывается

Получить ссылку на текущий объект можно при помощи специального ключевого слова `self`:

```
 > self
=> main
 > self.class
=> Object
```

В глобальной области мы ссылаемся на специальный безымянный объект `main`. Это объект класса `Object`, и у него нет имени, но мы можем получить ссылку на него при помощи `self`.

При определении и вызове метода можно явно указывать объект-получатель (листинг 9.17).

> ***Замечание***
> Более детально свойства ключевого слова `self` рассматриваются в *главе 16*.

Листинг 9.17. Явное указание получателя объекта. Файл self.rb

```
def self.convert(value:, factor: 1000)
  value * factor
end

puts self.convert(value: 11) # 11000
```

Глобальные методы можно и нужно определять без использования объекта-получателя. Если какой-либо элемент является необязательным, он всегда опускается.

Другое дело, что получатель можно использовать для ограничения области действия метода. Например, можно создать объект класса `Object` и определить метод `convert` только для этого объекта (листинг 9.18).

Листинг 9.18. Синглетон-метод объекта класса `Object`. Файл singleton_method.rb

```
object = Object.new
another = Object.new

def object.convert(value:, factor: 1000)
  value * factor
end

puts object.convert(value: 11) # 11000
puts another.convert(value: 5) # undefined method `convert'
```

Когда метод определяется на объекте, он существует только для этого объекта. Попытка вызова такого метода для других объектов завершается неудачей.

Поскольку такие методы существуют только в одном экземпляре для единственного объекта, они называются *синглетон-методами* (более подробно их свойства и приемы работы с такими методами рассматриваются в *главе 14*).

При вызове метода весьма часто можно встретить проверку его существования в отношении объекта при помощи метода `respond_to?` (листинг 9.19).

Листинг 9.19. Проверка существования метода `respond_to?`. Файл respond_to.rb

```
...
puts object.convert(value: 11) if object.respond_to? :convert
puts another.convert(value: 5) if another.respond_to? :convert
```

Как упоминалось в *главе 7*, в отношении объектов `nil` можно использовать безопасный вызов при помощи оператора `&.` (листинг 9.20).

Листинг 9.20. Безопасный вызов методов оператором `&.`. Файл safe_call.rb

```
object = Object.new
another = nil
...
puts object&.convert(value: 11) # 11000
puts another&.convert(value: 5) # nil
```

Впрочем, увлекаться оператором `&.` не стоит, чем меньше он используется в программе — тем лучше.

9.5. Псевдонимы методов

Ruby предоставляет специальное ключевое слово `alias`, при помощи которого можно создавать для методов *псевдонимы*. Это ключевое слово принимает в качестве первого аргумента псевдоним, а в качестве второго — название существующего метода (листинг 9.21).

Листинг 9.21. Псевдонимы методов. Файл alias.rb

```
def convert(value:, factor: 1000)
  value * factor
end

alias kg_to_gramms convert
alias kb_to_bytes convert

puts kg_to_gramms(value: 5)               # 5000
puts kb_to_bytes(value: 11, factor: 1024) # 11264
```

Следует обратить внимание, что `alias` — это ключевое слово, не метод. Поэтому между первым и вторым аргументами запятая не ставится.

9.6. Удаление метода

При помощи ключевого слова `undef` можно удалить метод. Попытки обращения к такому методу будут приводить к сообщению об ошибке — как будто бы метод не определялся вообще (листинг 9.22).

Листинг 9.22. Удаление метода. Файл undef.rb

```
def convert(value:, factor: 1000)
  value * factor
end

puts convert(value: 5) # 5000

undef convert

puts convert(value: 5) # undefined method `convert'
```

9.7. Рекурсивные методы

Внутри метода можно вызывать другие методы. Простейший пример приводится в листинге 9.23, где метод `good_format` в ходе работы вызывает метод `convert`.

Листинг 9.23. Вызов метода другим методом. Файл call_another_method.rb

```
def convert(value:, factor: 1000)
  value * factor
end

def good_format(weight, price)
  wt = format('Вес товарной позиции: %d', convert(value: weight))
  pr = format('Цена: %0.2f', price)
  {weight: wt, price: pr}
end

good = good_format(12, 5600)
puts good[:weight] # Вес товарной позиции: 12000
puts good[:price]  # Цена: 5600.00
```

Что произойдет, если метод будет вызывать сам себя? В Ruby это допустимая операция. Такие методы называются *рекурсивными*. Они часто используются для обхода деревьев, например, файловой системы. Продемонстрируем использование таких методов на примере задачи поиска факториала числа (рис. 9.2).

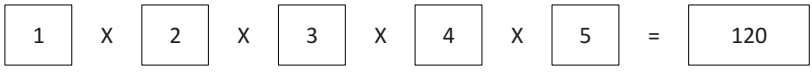


Рис. 9.2. Факториал числа 5!

Факториал числа — это все числа в диапазоне от 1 до значения числа, умноженные друг на друга. В листинге 9.24 приводится попытка создания рекурсивного метода `factorial`, который умножает число `number` на значение факториала от числа на единицу меньше.

Листинг 9.24. Попытка вычисления факториала. Файл recursive_wrong.rb

```
def factorial(number)
  number * factorial(number - 1)
end

puts factorial(5)
```

Попытка запуска программы из листинга 9.24 завершается неудачей:

```
$ ruby recursive_wrong.rb
Traceback (most recent call last):
       10920: from recursive_wrong.rb:5:in `<main>'
       10919: from recursive_wrong.rb:2:in `factorial'
       10918: from recursive_wrong.rb:2:in `factorial'
       10917: from recursive_wrong.rb:2:in `factorial'
       10916: from recursive_wrong.rb:2:in `factorial'
```

```
        10915: from recursive_wrong.rb:2:in `factorial'
        10914: from recursive_wrong.rb:2:in `factorial'
        10913: from recursive_wrong.rb:2:in `factorial'
         ... 10908 levels...
            4: from recursive_wrong.rb:2:in `factorial'
            3: from recursive_wrong.rb:2:in `factorial'
            2: from recursive_wrong.rb:2:in `factorial'
            1: from recursive_wrong.rb:2:in `factorial'
recursive_wrong.rb:2:in `factorial': stack level too deep (SystemStackError)
```

Возникает ошибка, сообщающая о переполнении стека, фактически же произошло зацикливание выполнения программы после почти 11 000 вызовов метода `factorial`.

Для того чтобы написать рабочий вариант рекурсии, необходимо разобраться в принципах ее работы (рис. 9.3).

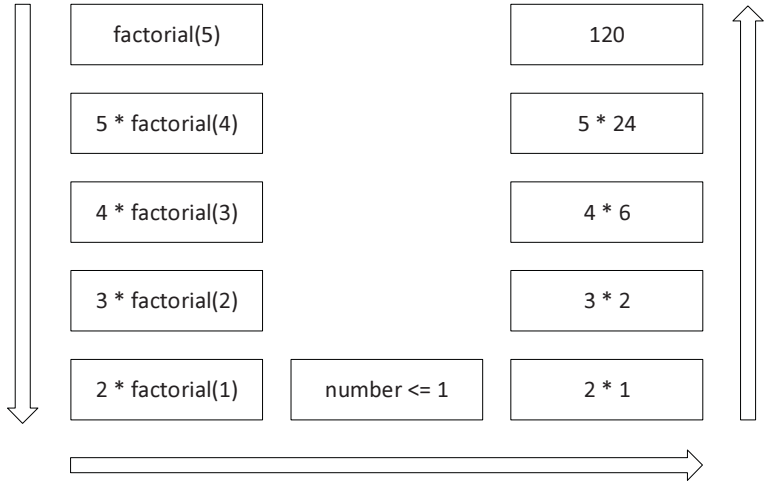


Рис. 9.3. Принцип работы рекурсии

Вызов метода `factorial(5)` приводит к вычислению выражения `5 * factorial(4)`. Вызов `factorial(4)` сводится к `4 * factorial(3)`, что в свою очередь приводит к `3 * factorial(2)`. Наконец, для вычисления `factorial(2)` потребуется выполнить выражение `2 * factorial(1)`. Если здесь не остановить рекурсию, программа сваливается в бесконечное вычисление `1 * factorial(0)`, `0 * factorial(-1)`, `-1 * factorial(-2)` и т. д.

Для остановки рекурсии необходимо создать точку возврата, чтобы рекурсия начала восхождение обратно. На рис. 9.3 процесс слева называется *спуском рекурсии*, подъем справа — *хвостом рекурсии*.

В качестве условия возврата можно всегда возвращать единицу — для вызова метода `factorial` с аргументом меньше или равным единицы (листинг 9.25).

Листинг 9.25. Вычисление факториала с использованием рекурсии. Файл recursive.rb

```
def factorial(number)
  return 1 if number <= 1
  number * factorial(number - 1)
end

puts factorial(5) # 120
```

9.8. Предопределенные методы

Не все методы необходимо разрабатывать самостоятельно. Ruby представляет большое количество готовых глобальных методов. Часть из них нам уже хорошо знакома: при помощи методов `puts`, `p` и `print` на протяжении предыдущих глав выводилась информация в стандартный поток вывода. В *главе 5* при помощи методов `local_variables`, `global_variables`, `instance_variables` и `class_variables` извлекались локальные, глобальные переменные, а также инстанс-переменные и переменные класса. С использованием методов `require` и `require_relative` в программу подключались файлы с Ruby-кодом и библиотеки (см. *главу 6*). Методы `sprintf` и `format` позволяют форматировать строки (см. *главу 7*).

Все эти методы сосредоточены в модуле `Kernel`, роль которого более подробно рассматривается в *разд. 20.6*. Помимо уже изученных методов, модуль `Kernel` предоставляет большое количество других глобальных методов, часть из которых представлена в последующих разделах.

9.8.1. Чтение входного потока

Информацию можно не только выводить в стандартный поток вывода, но и принимать ее от пользователя из стандартного потока ввода. Для этого предназначен метод `gets` (листинг 9.26).

Листинг 9.26. Получение информации от пользователя. Файл gets.rb

```
val = gets
```

Если запустить программу, она остановится и будет ожидать пользовательского ввода. Завершить ввод можно путем нажатия на клавишу <Enter>:

```
$ ruby gets.rb
Hello world!
```

Такой способ ввода данных не очень очевиден, поскольку весьма трудно догадаться, чего ожидает программа. Поэтому перед вызовом метода `gets` принято выводить поясняющий текст, сообщающий, что ожидается от пользователя (листинг 9.27).

Листинг 9.27. Дополнительный поясняющий вывод. Файл gets_puts.rb

```
puts 'Введите, пожалуйста, число '
val = gets
p val
```

Запуск программы приводит к следующему результату:

```
$ ruby gets_puts.rb
Введите, пожалуйста, число
235
"235\n"
```

Результат, который возвращает метод `gets`, является строкой. Причем перевод строки `\n` от нажатия клавиши <Enter> тоже в нее входит.

Для того чтобы преобразовать строку в число, можно воспользоваться методом `to_i`. Метод `puts` также можно заменить на `print` — чтобы позиция ввода оставалась на той же строке (листинг 9.28).

Листинг 9.28. Приведение результата ввода к целому числу. Файл gets_integer.rb

```
print 'Введите, пожалуйста, число '
val = gets.to_i
p val
```

Результат выполнения программы из листинга 9.28 может выглядеть следующим образом:

```
$ ruby gets_integer.rb
Введите, пожалуйста, число 235
235
```

В том случае, если результатом ввода является строка, избавиться от перевода в конце строки можно при помощи метода `chomp` (листинг 9.29).

Листинг 9.29. Использование метода chomp. Файл chomp.rb

```
print 'Введите, пожалуйста, строку '
val = gets.chomp
p val
```

Далее приводится возможный результат работы программы:

```
$ ruby chomp.rb
Введите, пожалуйста, строку Hello, world!
"Hello, world!"
```

Операция очистки строки от перевода настолько частая, что, начиная с Ruby 2.4, в метод `gets` добавлен дополнительный параметр `chomp`. Установка его в значение `true` приводит к автоматическому вызову `chomp` (листинг 9.30).

Листинг 9.30. Автоматическая очистка вводимых данных. Файл gets_chomp.rb

```
print 'Введите, пожалуйста, строку '
val = gets(chomp: true)
p val
```

Стандартный потоком ввода можно управлять, для чего используются константа `STDIN` и глобальная переменная `$stdin`. Например, можно запросить имя файла, открыть его при помощи метода `open` класса `File` (см. *главу 27*) и назначить в качестве источника ввода содержимое файла (листинг 9.31).

Листинг 9.31. Чтение из файла. Файл stdin.rb

```
print 'Введите, пожалуйста, имя файла '
filename = gets.chomp

$stdin = File.open(filename)

puts gets('$$')
```

Методу `gets`, который читает информацию из файла, передается необязательный аргумент — разделитель строки. В данном случае это последовательность `'$$'`, которая гарантированно не встречается в файле. В результате этого метод читает все содержимое файла, а не первую строку. По умолчанию метод `gets` ожидает перевода строки и, обнаружив ее в конце первой строки, прекращает дальнейшее чтение.

Если при выполнении программы ввести в качестве имени файла splat.rb, можно получить содержимое листинга 9.6:

```
$ ruby stdin.rb
Введите, пожалуйста, имя файла splat.rb
def multi_params(*params)
  p params # [1, 2, 3, 4]
end

multi_params(1, 2, 3, 4)
```

9.8.2. Остановка программы

Для остановки программы предназначен метод `abort` (листинг 9.32).

Листинг 9.32. Остановка программы. Файл abort.rb

```
puts 'Эта строка выводится.'
abort
puts 'А эта уже нет!'
```

В листинге 9.32 будет выведена первая строка, однако до последнего вызова `puts` дело не дойдет — работа программы будет остановлена методом `abort`. Метод

может принимать необязательный строковый параметр, который отправляется в стандартный поток ошибок STDERR (листинг 9.33).

Листинг 9.33. Вывод сообщения в стандартный поток ошибок. Файл abort_message.rb

```
puts 'Эта строка выводится.'
abort 'Аварийная остановка программы'
puts 'А эта уже нет!'
```

При вызове метода abort программа не сразу прекращает выполнение работы. При помощи метода at_exit можно задать выражения, которые безусловно выполняются перед завершением программы (листинг 9.34). В программе допускается несколько вызовов at_exit.

> ***Замечание***
>
> Для задания выражений в методе at_exit используется блок, работа с которым более детально рассматривается в *главе 12*.

Листинг 9.34. Использование метода at_exit. Файл at_exit.rb

```
at_exit { puts 'Завершение программы' }

puts 'Эта строка выводится.'
abort 'Аварийная остановка программы'
puts 'А эта уже нет!'
```

Блок кода, заданный at_exit, безусловно выполняется вне зависимости от того, завершается программа штатно или через abort. Результатом выполнения программы будут следующие строки кода:

```
$ ruby at_exit.rb
Эта строка выводится.
Аварийная остановка программы
Завершение программы
```

Схожим образом действует метод exit, который, как и abort, прекращает работу программы, выполняя зарегистрированные в at_exit выражения. В случае, если необходимо немедленно остановить работу программы, не выполняя at_exit-блоки, прибегают к методу exit! (листинг 9.35).

Листинг 9.35. Безусловная остановка программы методом exit!. Файл exit.rb

```
at_exit { puts 'Завершение программы' }

puts 'Эта строка выводится.'
exit!
puts 'А эта уже нет!'
```

В результате выполнения программы из листинга 9.35 будет выведена лишь одна строка:

```
Эта строка выводится.
```

Иногда необходимо лишь приостановить программу, не завершая ее работы. В этом случае прибегают к методу `sleep`, который приостанавливает работу программы на количество секунд, которые задаются через единственный параметр (листинг 9.36).

Листинг 9.36. Приостановка работы программы методом `sleep`. Файл sleep.rb

```
puts 'Подождем 5 секунд'
sleep 5
puts 'А теперь еще 3.5 секунды'
sleep 3.5
puts 'Работа программы завершена'
```

Метод `sleep` может принимать как целое, так и дробное количество секунд.

9.8.3. Методы-конструкторы

Ruby предоставляет ряд методов, которые начинаются с заглавной буквы. Следующие специальные методы нарушают современное соглашение об именовании методов, которые должны начинаться с маленькой (строчной) буквы:

- `Integer` — создание целого числа;
- `Float` — создание вещественного числа;
- `String` — создание строки;
- `Array` — создание массива;
- `Hash` — создание хэша.

В их использовании обычно нет надобности, т. к. синтаксические конструкции короче, читабельнее и нагляднее:

```
 > var = String('hello')
=> "hello"
 > var = 'hello'
=> "hello"
 > number = Integer(10)
=> 10
 > number = 10
=> 10
```

Тем не менее, в старых проектах и гемах можно обнаружить иногда использование методов `Array` и `Hash`. Как правило, для всех операций, ради которых можно было бы задействовать эти методы, имеется современная и более короткая реализация.

Метод `Array` осуществляет поиск в объекте методов `to_ary`, затем `to_a`. Обнаружив один из них — возвращает результирующий массив. В случае, если обнаружить эти методы не удается, объект сам становится единственным элементом массива:

```
 > Array(1..5)
=> [1, 2, 3, 4, 5]
 > Array('hello, world!')
=> ["hello, world!"]
```

Однако на практике короче и элегантнее выглядит код, в котором у объекта явно вызываются методы `to_ary` или `to_a`:

```
 > (1..5).to_a
=> [1, 2, 3, 4, 5]
 > ['hello, world!']
=> ["hello, world!"]
```

9.9. Логические методы

Методы, которые завершаются вопросительным знаком, называются *логическими методами*. Добавление вопросительного знака в название не изменяет свойств методов — это лишь сигнал другим разработчикам о том, что метод возвращает истинное (`true`) или ложное (`false`) значения.

Такие методы, как правило, используются либо в конструкциях ветвления, либо в логических выражениях. В предыдущих главах мы уже встречались с логическими методами, которые предоставляет язык Ruby (табл. 9.1).

Таблица 9.1. *Логические методы*

Оператор	Описание
`belongs_to?`	Проверяет, может ли к объекту быть применен метод
`even?`	Проверяет четность числа
`odd?`	Проверяет нечетность числа
`finit?`	Проверяет, является ли число конечным
`infinit?`	Проверяет, является ли число бесконечным
`nan?`	Проверяет, является ли число допустимым
`nil?`	Проверяет, содержит ли переменная в качестве значения объект `nil`
`zero?`	Проверяет, является ли число равным `0`
`positive?`	Проверяет, больше ли число нуля
`negative?`	Проверяет, меньше ли число нуля

В табл. 9.1 представлен далеко не полный список логических методов, однако всех их объединяет знак вопроса на конце и возврат логического значения.

Логические методы можно создавать самостоятельно. Соглашения Ruby-сообщества требуют добавлять вопросительный знак в название метода, если он возвращает `true` или `false`.

В листинге 9.37 приводится пример программы, которая запрашивает у пользователя число и сообщает, входит ли значение в диапазон чисел от 1 до 10.

Листинг 9.37. Использование логических методов. Файл logic.rb

```
def right?(num)
  (1..10).include? num
end

print 'Пожалуйста, введите число '
number = gets.to_i

if right?(number)
  puts 'Число входит в диапазон от 1 до 10'
else
  puts 'Число не входит в диапазон от 1 до 10'
end
```

Для организации программы создается метод right?, который проверяет корректность условия. Этот метод задействуется в конструкции if для того, чтобы выбрать фразу для вывода.

9.10. bang-методы

В конце названия метода может быть указан восклицательный знак. Такие методы называются *bang-методами*. Их следует применять с оглядкой, поскольку они, как правило, изменяют состояние объекта или совершают какие-то другие «опасные» действия.

Методы с восклицательным знаком на конце названия часто имеют пару. Например, для получения строки в верхнем режиме можно использовать метод upcase (листинг 9.38).

Листинг 9.38. Использование методов upcase и upcase!. Файл upcase.rb

```
hello = 'Hello, world!'

puts hello.upcase  # HELLO, WORLD!
puts hello         # Hello, world!

puts hello.upcase! # HELLO, WORLD!
puts hello         # HELLO, WORLD!
```

Метод upcase не затрагивает оригинальную строку, вместо этого возвращается новый объект-строка. Для этого метода существует bang-вариант: upcase!, использование которого приводит к изменению исходной строки.

Можно самостоятельно создавать bang-методы, причем не обязательно, чтобы такие методы были парными. Восклицательным знаком можно пометить любую потенциально-опасную необратимую операцию: удаление файла, изменение состояния базы данных, аварийная остановка программы и т. д.

Задания

1. Создайте метод `sum`, который принимает любое количество числовых аргументов и возвращает их сумму.
2. Создайте метод, который проверяет, является ли текущий год високосным. В случае, если год високосный, метод должен возвращать `true` (истину), иначе должна возвращаться `false` (ложь).
3. Запросите у пользователя название цвета и верните номер цвета из следующей таблицы:
   - 1 — красный;
   - 2 — оранжевый;
   - 3 — желтый;
   - 4 — зеленый;
   - 5 — голубой;
   - 6 — синий;
   - 7 — фиолетовый.
4. Создайте класс пользователя `User`, объекты которого сохраняют фамилию, имя и отчество пользователя. Запросите в консоли при помощи метода `gets` данные для трех пользователей. С использованием полученных сведений создайте массив, содержащий три объекта класса `User`.
5. Создайте метод `cel2far` для перевода градусов Цельсия в градусы по Фаренгейту, а также обратный метод `fal2cel` — для перевода градусов по Фаренгейту в градусы Цельсия.
6. Числа Фибоначчи — это последовательность вида 0, 1, 1, 2, 3, 5, ..., где каждое число является суммой двух предыдущих чисел. Создайте скрипт, который бы вычислял любое наперед заданное число Фибоначчи.

ГЛАВА 10

Циклы

Файлы с исходными кодами этой главы находятся в каталоге *cycles* сопровождающего книгу электронного архива.

Повторяющиеся операции в программировании принято организовывать при помощи *циклов*. Для организации циклов Ruby предоставляет три ключевых слова: `while`, `until` и `for`. Циклы `while` и `until` очень похожи на ключевые слова `if` и `unless`. Они так же принимают в качестве аргумента логическое значение или выражение, по которому принимается решение о продолжении или остановке цикла. Цикл `for` предназначен для обхода коллекций и больше напоминает цикл `foreach` или `for in` из других языков программирования, нежели классический `for` из C-подобных языков программирования.

На практике циклы практически не используют — большое распространение получили *итераторы*, которые рассматриваются в *главе 11*. Несмотря на то, что в современных проектах часто можно не найти ни одного цикла, они все еще задействуются для создания собственных итераторов.

10.1. Цикл *while*

Ключевое слово `while` принимает в качестве единственного параметра логическое выражение, которое называется *условием цикла*. Завершается цикл ключевым словом `end`. Все, что расположено между условием и `end`, является телом цикла. В теле цикла можно размещать любой Ruby-код, который будет повторяться на каждой итерации.

В листинге 10.1 в качестве условия цикла задается «истина» — объект `true`. Так как в ходе работы программы объект `true` меняться не будет, цикл станет выполняться бесконечно, выводя фразу `Hello, world!`.

Листинг 10.1. Бесконечный цикл `while`. Файл while.rb

```
while true
  puts 'Hello, world!'
end
```

Прервать работу программы можно, воспользовавшись комбинацией клавиш <Ctrl>+<C>. Если речь идет не о сервере или графической программе, создание бесконечных циклов редко требуется на практике и чаще рассматривается как ошибка.

Чаще всего условие цикла формируется таким образом, чтобы на первых итерациях оно было истинным, а рано или поздно становилось ложным, и цикл прекращал бы свою работу.

В листинге 10.2 у пользователя при помощи метода `gets` запрашивается, сколько раз необходимо вывести фразу `'Hello, world!'`. Кроме того, заводится переменная `i`, значение которой увеличивается на единицу на каждой итерации цикла, чтобы условие цикла со временем стало ложным.

Листинг 10.2. Конечный цикл `while`. Файл while_gets.rb

```
print 'Пожалуйста, введите количество повторов: '
max_iterates = gets.to_i
i = 0

while i < max_iterates
  puts 'Hello, world!'
  i += 1
end
```

Запуск программы может выглядеть следующим образом:

```
$ ruby while_gets.rb
Пожалуйста, введите количество повторов: 3
Hello, world!
Hello, world!
Hello, world!
```

После того как пользователь вводит цифру `3`, она попадает в переменную `max_iterates`, значение переменной счетчика при этом равно `0` (рис. 10.1).

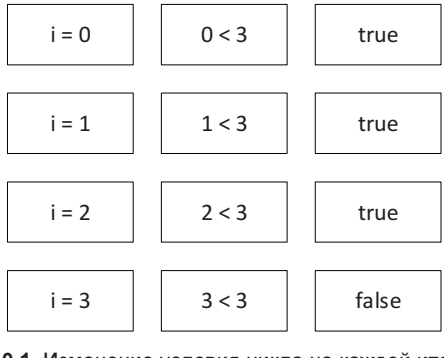


Рис. 10.1. Изменение условия цикла на каждой итерации

Так как сравнение `0 < 3` возвращает «истину» `true`, цикл выполняет тело, увеличивая последним выражением значение счетчика `i` на единицу. После выполнения цикла вновь происходит вычисление условия в ключевом слове `while`. Условие `1 < 3` также истинно, и цикл второй раз выполняет тело, в конце которого переменная `i` получает значение 2. Снова происходит сравнение `2 < 3`, которое опять является истинным. Цикл выводит фразу `'Hello, world!'` в третий раз и увеличивает значение `i` до 3. На этот раз условие `3 < 3` оказывается ложным, и цикл прекращает свою работу. Интерпретатор переходит за ключевое слово `end` и приступает к выполнению выражений, следующих за циклом.

Логика построения условия позволяет нам не обрабатывать ситуацию ввода пользователем нулевого или отрицательного значения. Условие `0 < 0` или `0 < -5` сразу будет ложным, и цикл не выполнит ни одной итерации:

```
$ ruby while_gets.rb
Пожалуйста, введите количество повторов: 0
$ ruby while_gets.rb
Пожалуйста, введите количество повторов: -5
```

После условия цикла может использоваться необязательное ключевое слово `do` (листинг 10.3).

Листинг 10.3. Использование ключевого слова do. Файл while_do.rb

```
...
while i < max_iterates do
  puts 'Hello, world!'
  i += 1
end
```

В такой форме тело цикла очень напоминает Ruby-конструкцию *блок* (см. *главу 12*). Тем не менее это именно синтаксическая конструкция, не имеющая к блокам Ruby никакого отношения. Чтобы код был короче, компактнее и не возникало ненужных ассоциаций с блоками, ключевое слово `do` всегда опускается. Исключение могут составлять однострочные выражения — например, при составлении их в интерактивном Ruby (`irb`):

```
 > n = 0; while n < 3 do puts n; n += 1 end
0
1
2
=> nil
```

Если в этом случае не использовать ключевое слово `do`, интерпретатор Ruby «запутается» и не сможет определить, где заканчивается условие, а где начинается тело цикла.

В программах, расположенных в RB-файлах нет необходимости располагать выражения в одной строке. В том случае, если тело цикла состоит из одного выражения,

чаще используется `while`-модификатор, который располагается после выражения (листинг 10.4).

Листинг 10.4. Использование `while` в режиме модификатора. Файл while_modifier.rb

```
print 'Пожалуйста, введите количество повторов: '
max_iterates = gets.to_i
i = 0

puts 'Hello, world!' while (i += 1) <= max_iterates
```

В приведенном примере значение переменной-счетчика `i` увеличивается прямо в условии `while`. В результате в теле цикла остается только одно выражение для вывода строки в стандартный поток вывода. Это позволяет сократить цикл до одной строки, разместив `while` в конце выражения.

Условия можно размещать не только в начале, но и в конце цикла. Для этого цикл начинается с ключевого слова `begin`, которое обозначает начало цикла, при этом ключевое слово `while` и условие размещаются в конце после ключевого слова `end` (листинг 10.5).

Листинг 10.5. Размещение условия в конце цикла. Файл begin_while.rb

```
...
begin
  puts 'Hello, world!'
  i += 1
end while i < max_iterates
```

В этом случае тело цикла будет выполнено хотя бы один раз, т. к. условие продолжения цикла будет проверено лишь в конце итерации:

```
$ ruby begin_while.rb
Пожалуйста, введите количество повторов: 0
Hello, world!
```

Когда цикл совершается в глобальной области видимости, мы уже находимся внутри объекта класса `Object`. Поэтому в циклах допускается использование инстанс-переменных (листинг 10.6).

Листинг 10.6. Использование инстанс-переменных в цикле. Файл while_instance.rb

```
print 'Пожалуйста, введите количество повторов: '
@max_iterates = gets.to_i
@i = 0

while @i < @max_iterates
  puts @i
  @i += 1
end
```

Хотя приведенный пример рабочий, нет никакой пользы от использования инстанс-переменных. Область действия переменных расширяется. В текущей реализации надобности в этом нет, т. к., в отличие от методов, конструкция `while` не является барьером для локальных переменных.

Однако инстанс-переменные можно использовать внутри глобальных методов. Цикл из листинга 10.6 можно переписать таким образом, чтобы условие цикла (увеличение переменной-счетчика) обслуживалось методом (листинг 10.7).

> ***Замечание***
>
> Так как метод `condition` из листинга 10.7 возвращает только два состояния: «ложь» и «истина», согласно соглашениям, принятым в Ruby-сообществе, он должен завершаться вопросительным знаком `condition?`. Более подробно методы, завершающиеся специальными символами `?`, `!` и `=`, рассматриваются в *главах 9* и *14*.

Листинг 10.7. Файл while_condition.rb

```
def condition
  @i ||= 0
  @max_iterates ||= begin
    print 'Пожалуйста, введите количество повторов: '
    gets.to_i
  end

  @i += 1
  @i <= @max_iterates
end

puts @i while condition
```

Вся логика манипуляции с инстанс-переменными `@i` и `@max_iterates` вынесена в метод `condition`. Это позволило упростить цикл `while` до одной строки с `while`-модификатором.

В силу того, что метод `condition` используется в качестве аргумента `while`, он должен возвращать либо `true`, либо `false`. Поэтому условие сравнения счетчика `@i` и предельного значения `@max_iterates` располагается на последней строке метода. Так как операция увеличения счетчика теперь располагается до условия, пришлось изменить оператор < на <=.

Инициализация переменных `@i` и `@max_iterates` выполнена при помощи оператора `||=` (см. *главу 8*). Так как инстанс-переменные хранятся в объекте, они сохраняют состояние, сколько бы раз ни вызывался метод `condition`. Лишь при первом вызове значение этих переменных будет установлено в `nil`. Таким образом, при первом вызове метода переменной `@i` будет задано значение `0`, а `@max_iterates` — значение, введенное пользователем. При всех последующих вызовах значение переменным уже окажется установленным, и оператор `||=` будет оставлять значение переменных без изменений.

10.2. Вложенные циклы

Допускается вложение циклов друг в друга. В листинге 10.8 при помощи метода `gets` у пользователя запрашивается число и при помощи вложенного цикла `while` выводится таблица умножения. Результатом работы программы должна стать квадратная таблица, по горизонтали и вертикали которой расположены значения, возрастающие от единицы до максимального. На пересечениях строк и столбцов — результат перемножения двух чисел.

Листинг 10.8. Вывод таблицы умножения. Файл multiplication_table.rb

```
print 'Пожалуйста, введите максимальное значение: '
max_iterates = gets.to_i
i = 1

size = (max_iterates * max_iterates).to_s.size + 1

while i <= max_iterates
  j = 1
  while j <= max_iterates
    print format("% #{size}d ", i * j)
    j += 1
  end
  i += 1
  puts
end
```

Под число следует отвести достаточно места, поэтому необходимо вычислить максимальное значение для строки форматирования в методе `format`:

```
 > format('% 3d', 12)
=> " 12"
 > format('% 4d', 122)
=> " 122"
 > format('% 5d', 1024)
=> " 1024"
```

Для этого вычисляется максимально возможное значение в таблице: квадрат введенного пользователем числа: `max_iterates * max_iterates`. При помощи метода `to_s` оно преобразуется в строку, количество символов которой вычисляется при помощи метода `size`.

Первый цикл `while` несет ответственность за формирование строк. На каждой итерации этого цикла выполняется дополнительный `while`-цикл с переменной-счетчиком `j`. Этот цикл несет ответственность за формирование значений в рамках строки (рис. 10.2).

Именно поэтому во внутреннем цикле `while` используется метод `print`, чтобы неявными переводами строк не разрушать формируемую таблицу. В конце итерации

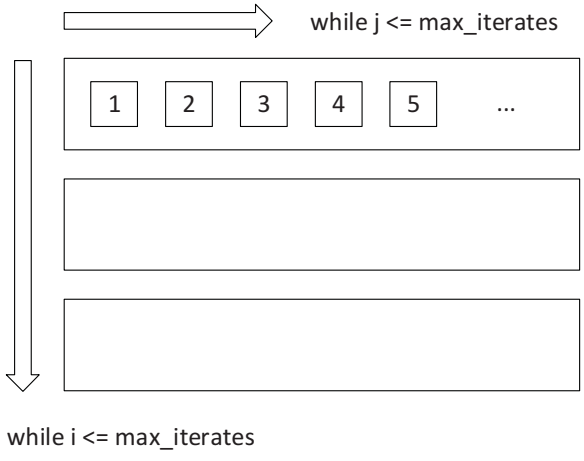


Рис. 10.2. Ответственность циклов `while`

внешнего цикла перевод строки добавляется путем вызова метода `puts` без аргументов.

Вызов программы с последующим вводом числа `14` приводит к формированию следующей таблицы:

```
$ ruby multiplication_table.rb
Пожалуйста, введите максимальное значение: 14
   1    2    3    4    5    6    7    8    9   10   11   12   13   14
   2    4    6    8   10   12   14   16   18   20   22   24   26   28
   3    6    9   12   15   18   21   24   27   30   33   36   39   42
   4    8   12   16   20   24   28   32   36   40   44   48   52   56
   5   10   15   20   25   30   35   40   45   50   55   60   65   70
   6   12   18   24   30   36   42   48   54   60   66   72   78   84
   7   14   21   28   35   42   49   56   63   70   77   84   91   98
   8   16   24   32   40   48   56   64   72   80   88   96  104  112
   9   18   27   36   45   54   63   72   81   90   99  108  117  126
  10   20   30   40   50   60   70   80   90  100  110  120  130  140
  11   22   33   44   55   66   77   88   99  110  121  132  143  154
  12   24   36   48   60   72   84   96  108  120  132  144  156  168
  13   26   39   52   65   78   91  104  117  130  143  156  169  182
  14   28   42   56   70   84   98  112  126  140  154  168  182  196
```

10.3. Досрочное прекращение циклов

Существует несколько способов досрочного прекращения работы циклов. Одним из основных способов при этом является использование ключевого слова `break`. Как правило, вызов ключевого слова происходит по условию с участием `if` или `unless`. В листинге 10.9 вводится ограничение на выполнение цикла более пяти раз. Если пользователь укажет большее значение, сработает `if`-условие, и цикл будет досрочно завершен ключевым словом `break`.

Листинг 10.9. Прерывание цикла при помощи `break`. Файл break.rb

```
print 'Пожалуйста, введите количество повторов: '
max_iterates = gets.to_i
i = 0

while i < max_iterates do
  puts 'Hello, world!'
  i += 1
  break if i >= 5
end
```

Прерывать можно не только цикл целиком, но и текущую итерацию, — для этого предназначено ключевое слово `next`. В листинге 10.10 у пользователя запрашивается числовое значение, и в цикле выводится каждый нечетный элемент от `1` до введенного значения. Для того чтобы не получить бесконечный цикл, операцию увеличения счетчика `i` необходимо разместить до возможного вызова `next`. Иначе переход в начало цикла без изменения `i` будет приводить к тому, что условие `i < max_iterates` продолжит оставаться истинным, и цикл станет выполняться вечно.

> ***Замечание***
>
> В C-подобных языках вместо ключевого слова `next` используется `continue`.

Листинг 10.10. Прерывание итерации цикла при помощи `next`. Файл next.rb

```
print 'Пожалуйста, введите число: '
max_iterates = gets.to_i
i = 0

while i < max_iterates do
  i += 1
  next if i.even?
  puts i
end
```

В цикле пропускаются все четные значения при помощи метода `even?`, который возвращает `true` — для четного числа и `false` — для нечетного:

```
 > 5.even?
=> false
 > 4.even?
=> true
```

Как только условие в `if`-модификаторе становится истинным, вызывается ключевое слово `next`, которое прерывает текущую итерацию, и интерпретатор перемещается в начало цикла к ключевому слову `while`. Нечетные числа выводятся, однако на

итерациях с четными числами до выражения puts дело не доходит. Далее приводится вывод программы в том случае, если пользователь выбирает число 10:

```
$ ruby next.rb
Пожалуйста, введите число: 10
1
3
5
7
9
```

В противовес ключевому слову next, который прерывает текущую итерацию, Ruby предоставляет ключевое слово redo, задача которого заключается в повторении итерации. В листинг 10.11 приводится пример программы, где на каждой итерации вызывается метод rand(0..1), выдающий по случайному закону либо 0, либо 1. В том случае, если метод возвращает значение 0, вызывается ключевое слово redo. Для проверки равенству нулю используется метод zero?.

Листинг 10.11. Повторение итерации цикла при помощи redo. Файл redo.rb

```
print 'Пожалуйста, введите число: '
max_iterates = gets.to_i
i = 0

while i < max_iterates do
  puts i
  redo if rand(0..1).zero?
  i += 1
end
```

Так как метод rand, вызванный подряд, может выдать целую серию нулей, окажется, что одна и та же цифра выводится несколько раз. Возможный вывод программы тогда будет выглядеть следующим образом:

```
$ ruby redo.rb
Пожалуйста, введите число: 3
0
0
1
1
1
2
```

10.4. Цикл *until*

Цикл until, как и while, предназначен для выполнения итераций. Однако условие для продолжения выполнения цикла должно быть ложным. Изменение его состояния на истинное приводит к завершению работы цикла (листинг 10.12).

Листинг 10.12. Использование цикла `until`. Файл until.rb

```
print 'Пожалуйста, введите количество повторов: '
max_iterates = gets.to_i
i = 0

until i >= max_iterates
  puts 'Hello, world!'
  i += 1
end
```

Запуск программы из листинга 10.12 приводит к следующему результату:

```
$ ruby until.rb
Пожалуйста, введите количество повторов: 3
Hello, world!
Hello, world!
Hello, world!
```

На практике стараются не злоупотреблять циклом `until`, т. к. он работает с условиями отрицания. Считается, что отрицание воспринимается сложнее, чем положительные утверждения.

Цикл `until` обладает всеми свойствами, которые присущи `while`. Поддерживается необязательное ключевое слово `do`, модифицированная форма, расположение условия после ключевого слова `end`, а также использование ключевых слов `break` и `next` для досрочного прерывания цикла и итерации.

10.5. Цикл *for*

Цикл `for` предназначен для обхода коллекций и итераторов. В листинге 10.13 цикл `for` обходит элементы массива `[1, 2, 3, 4]`. На каждой итерации переменная `x` получает очередной элемент массива. В теле цикла переменная выводится в стандартный поток вывода при помощи метода `puts`.

Листинг 10.13. Использование цикла `for`. Файл for.rb

```
for x in [1, 2, 3, 4]
  puts x
end
```

При помощи цикла `for` можно обходить не только массивы. В листинге 10.14 приводится пример обхода диапазона.

Листинг 10.14. Обход диапазона при помощи цикла `for`. Файл for_range.rb

```
for x in 1..4
  puts x
end
```

В отличие от `while` и `until`, не допускается размещение ключевого слова `for` после `end`. Не допускается использование цикла `for` в качестве модификатора. Однако поддерживаются все остальные элементы: необязательное ключевое слово `do`, использование ключевых слов `break` и `next`.

Задания

1. Создайте программу, которая запрашивает два целых числа. Выведите результат деления чисел в консоль. Для того чтобы предотвратить ошибку при делении на ноль, убедитесь, что второе число не нулевое. В цикле запрашивайте второе число до тех пор, пока пользователь не введет число, отличное от нуля.
2. Создайте массив из 10 элементов со случайными числами от 0 до 99. Найдите максимальное и минимальное значение этого массива.
3. Создайте массив из 10 строк с фамилиями пользователей: Иванов, Петров, Сидоров, Алексеева, Казанцев, Антропов, Анисимова, Кузнецов, Соловьев, Кошкина. Выведите список пользователей в алфавитном порядке.
4. Создайте программу, которая выводит интервал дней текущей недели: `'07.10.2019-13.10.2019'`.
5. В UNIX-подобных операционных системах имеется утилита `ncal`, которая выводит календарь на текущий месяц:

```
$ ncal
    Февраль 2019
пн     4 11 18 25
вт     5 12 19 26
ср     6 13 20 27
чт     7 14 21 28
пт  1  8 15 22
сб  2  9 16 23
вс  3 10 17 24
```

   Создайте на Ruby программу, которая выводит аналогичный календарь для текущего месяца.

ГЛАВА 11

Итераторы

Файлы с исходными кодами этой главы находятся в каталоге *iterators* сопровождающего книгу электронного архива.

Итераторы — это краеугольный камень языка Ruby. Практически все операции, требующие повторения, программируются не с помощью циклов, а на основе итераторов. На практике можно часто встретить Ruby-проекты, где нет ни одного цикла.

В этой главе представлены итераторы и блоки, без которых итераторы невозможно представить. Мы рассмотрим итераторы: `loop`, `each`, `times`, `upto`, `downto` и `tap`.

Кроме того, познакомимся с итераторами коллекций, с которыми должен быть знаком каждый рубист: `each`, `map`, `select`, `reject` и `reduce`.

11.1. Итераторы и блоки

Итераторы, так же как и циклы (см. *главу 10*), позволяют многократно выполнять набор Ruby-выражений. Но, в отличие от `while`, `until` и `for`, итераторы не являются ключевыми словами — это методы.

Выражения, предназначенные для повторения в итераторе, помещаются в блок кода, который записывается либо в фигурных скобках, либо между ключевыми словами `do` и `end` (листинг 11.1).

Листинг 11.1. Бесконечный цикл. Файл loop.rb

```
loop { puts 'Hello, world!' }
loop do
  puts 'Hello, world!'
end
```

В листинге 11.1 на примере итератора `loop` показаны две формы блока, в котором размещается единственное Ruby-выражение, — вывод фразы `'Hello, world!'` в стандартный поток вывода.

Итератор `loop` бесконечно выполняет содержимое блока, и если его не остановить дополнительными средствами — например, ключевым словом `break`, программу придется останавливать комбинацией клавиш <Ctrl>+<C>.

При оформлении блоков действует соглашение:

- если содержимое представляет лишь одну строку, используются фигурные скобки:

```
loop { puts 'Hello, world!' }
```

- если в блоке необходимо разместить несколько выражений, используются ключевые слова `do` и `end`:

```
loop do
  puts 'Hello, world!'
  puts 'Hello, Ruby!'
end
```

11.2. Обход итераторами массивов и хэшей

Предыдущая глава завершилась рассмотрением цикла `for`. В листинге 11.2 представлен массив, состоящий из английских названий цветов радуги.

Листинг 11.2. Обход массива циклом `for`. Файл for.rb

```
rainbow = %w[red orange yellow green gray indigo violet]

for color in rainbow
  puts color
end
```

На каждой итерации очередной элемент массива помещается в переменную `color`, которая доступна в теле цикла.

Этот цикл можно заменить итератором — например, итератором `each` (листинг 11.3). Итератор `each` полностью эквивалентен циклу `for` — он пробегает от первого до последнего элемента массива. И на каждой итерации в блок передается очередной элемент массива.

Листинг 11.3. Обход массива итератором `each`. Файл each.rb

```
rainbow = %w[red orange yellow green gray indigo violet]

rainbow.each { |color| puts color }
```

Как видно из листинга 11.3, блоки могут принимать параметры, которые размещаются между двумя вертикальными чертами. В нашем случае в блок передается элемент массива для текущей итерации.

Итераторы более гибкие, чем циклы. Например, итератор `each` можно применить для обхода хэша (листинг 11.4).

Листинг 11.4. Обход хэша итератором `each`. Файл each_hash.rb

```
rainbow = {
  red: 'красный',
  orange: 'оранжевый',
  yellow: 'желтый',
  green: 'зеленый',
  blue: 'голубой',
  indigo: 'синий',
  violet: 'фиолетовый'
}

rainbow.each { |key, name| puts "#{key}: #{name}" }
```

Здесь в блок передаются два параметра: `key` — соответствующий ключу хэша и `name`, который соответствует значению. Каждый из этих параметров может быть использован внутри блока. В примере они интерполируются в строку, которая выводится при помощи метода `puts`.

Если запустить программу на выполнение, результатом будут следующие строки:

```
red: красный
orange: оранжевый
yellow: желтый
green: зеленый
blue: голубой
indigo: синий
violet: фиолетовый
```

11.3. Итератор *times*

Итераторы не обязательно должны обслуживать коллекции. Например, итератор `times` применяется к целым числам и позволяет выполнить блок указанное в числе количество раз (листинг 11.5).

Листинг 11.5. Использование итератора `times`. Файл times.rb

```
5.times { |i| puts i }
```

Запустив программу на выполнение, можно получить последовательный вывод значений от 0 до 4:

```
0
1
2
3
4
```

Для вещественного числа итератор `times` уже не сработает, Ruby-интерпретатор выведет сообщение об ошибке:

```
> (5.0).times { |i| puts i }
NoMethodError (undefined method `times' for 5.0:Float)
```

Итераторы — это методы, они могут быть реализованы в классе или нет. Если итератор не реализован, воспользоваться им не получится.

11.4. Итераторы *upto* и *downto*

Для итерирования от одного числа к другому применяется итератор `upto`. В листинге 11.6 итератор пробегает значения от 5 до 10.

Листинг 11.6. Использование итератора `upto`. Файл upto.rb

```
5.upto(10) { |i| puts i }
```

Итератор `downto` позволяет пробегать числа с шагом `-1` (листинг 11.7).

Листинг 11.7. Использование итератора `downto`. Файл downto.rb

```
10.downto(5) { |i| puts i }
```

Запуск программы из листинга 11.7 приводит к выводу чисел от 10 до 5:

```
10
9
8
7
6
5
```

11.5. Итераторы коллекций

Среди итераторов выделяют несколько, которые должен знать любой Ruby-разработчик, — это итераторы коллекций (табл. 11.1). Для многих из них существуют синонимы — то или иное имя используется в зависимости от семантики программы.

> ***Замечание***
>
> В табл. 11.1 приводится лишь часть итераторов модуля `Enumerable`. Кроме того, в этой главе рассматриваются лишь базовые свойства коллекций. Более детально работа с коллекциями рассмотрена в *главах 21, 23* и *24*.

Таблица 11.1. Итераторы коллекций

Итератор	Синоним
each	
each_with_index	
map	collect
select	find_all
reject	delete_if
reduce	inject
each_with_object	

11.5.1. Итератор *each*

Итератор `each` уже рассматривался ранее в этой главе — он выполняет блок столько раз, сколько элементов содержит коллекция (см. листинг 11.3). При этом на каждой итерации в блок передается один параметр — в случае массива, и два — если обходится хэш (см. листинг 11.4).

Итератор `each` не изменяет исходную коллекцию — если будет выведено возвращаемое значение, можно убедиться, что коллекция остается неизменной (листинг 11.8).

Листинг 11.8. Итератор `each` не изменяет исходную коллекцию. Файл each_return.rb

```
arr = [1, 2, 3, 4, 5]

result = arr.each { |x| puts x + 1 }

p result
```

Результатом выполнения программы из листинга 11.8 будут следующие строки:

```
$ ruby each_return.rb
2
3
4
5
6
[1, 2, 3, 4, 5]
```

Как можно видеть, в блоке выводятся значения массива, увеличенные на единицу, в то время как сам итератор возвращает ту же самую коллекцию, к которой он был применен.

11.5.2. Итератор *each_with_index*

В листинге 11.8 производится обход монотонно-возрастающего массива, и в нумерации элементов нет необходимости. В том же случае, когда осуществляется обход массива строк или символов, часто требуется пронумеровать элементы. Для этого удобно воспользоваться итератором `each_with_index`. Блок этого итератора принимает два параметра: первый — элемент коллекции, второй — текущий номер (листинг 11.9).

Листинг 11.9. Использование итератора `each_with_index`. Файл each_with_index.rb

```
rainbow = %w[red orange yellow green gray indigo violet]
rainbow.each_with_index { |color, i| puts "#{i}: #{color}" }
```

В приведенном примере в итераторе назначается индекс каждому из цветов радуги. Результатом работы программы будут следующие строки:

```
0: red
1: orange
2: yellow
3: green
4: gray
5: indigo
6: violet
```

11.5.3. Итератор *map*

Если необходим массив, содержащий преобразованные элементы коллекции, вместо `each` используется итератор `map`. Этот итератор возвращает новую коллекцию, количество элементов в которой совпадает с количеством элементов в исходной коллекции. Однако каждый элемент заменен на результат вычисления в блоке (листинг 11.10).

Листинг 11.10. Использование итератора `map`. Файл map.rb

```
result = [1, 2, 3, 4, 5].map { |x| x + 1 }
p result # [2, 3, 4, 5, 6]
```

Следует обратить внимание на то, что значение, которое возвращает блок, используется для формирования элементов нового массива. Поэтому, если оставить в блоке метод `puts`, элементы будут выведены в отдельных строках, но результирующий массив будет содержать только значения `nil` — результат, который возвращает метод `puts` (листинг 11.11).

Листинг 11.11. Файл map_puts.rb

```
result = [1, 2, 3, 4, 5].map { |x| puts x + 1 }
p result # [nil, nil, nil, nil, nil]
```

У итератора `map` имеется синоним — `collect` (листинг 11.12). Форма `collect` встречается в программах реже, `map` — более короткий и поэтому более популярен.

Листинг 11.12. `collect` является синонимом для итератора `map`. Файл collect.rb

```
p [1, 2, 3, 4, 5].collect { |x| x + 1 } # [2, 3, 4, 5, 6]
```

На начальных этапах обучения, пока нет навыка использования итераторов, возникает соблазн использовать какой-либо один итератор для решения всех циклических задач. Так, в листинге 11.13 массив `arr` заполняется при помощи итератора `each` — сначала создается пустой массив, который заполняется в итераторе `each` квадратами элементов.

Листинг 11.13. Некорректное заполнение массива. Файл each_incorrect.rb

```
arr = []
[1, 2, 3, 4, 5].each { |x| arr << x * x }
p arr # [1, 4, 9, 16, 25]
```

Код из приведенного примера можно сократить за счет использования итератора `map`, который как раз и предназначен для преобразования коллекции в другую коллекцию (листинг 11.14).

Листинг 11.14. Корректное заполнение массива. Файл map_correct.rb

```
arr = [1, 2, 3, 4, 5].map { |x| x * x }
p arr # [1, 4, 9, 16, 25]
```

11.5.4. Итераторы *select* и *reject*

Итератор `map` преобразует элементы коллекции, не изменяя количества элементов (рис. 11.1).

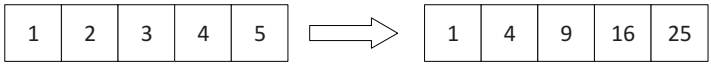


Рис. 11.1. Преобразование коллекций итератором `map`

Для того чтобы отфильтровать содержимое массива, потребуются отдельные итераторы: `select` и `reject` (рис. 11.2). Для этих итераторов также предусмотрены синонимы: `find_all` и `delete_if` соответственно.

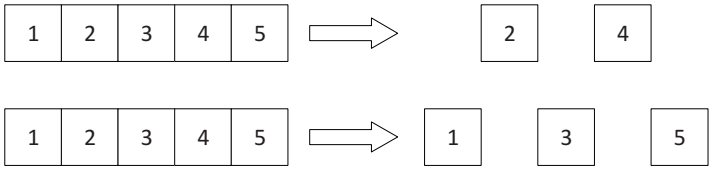


Рис. 11.2. Фильтрация коллекции итераторами `select` и `reject`

Блоки итератора `select` должны возвращать либо `true`, либо `false`. В зависимости от того, возвращает блок `true` или `false`, элемент старой коллекции либо попадает в новую, либо игнорируется. Таким способом можно фильтровать коллекцию, оставляя только часть элементов. Например, в листинге 11.15 извлекаются только четные элементы массива.

> ***Замечание***
>
> Следует помнить о том, что `nil` рассматривается как `false`, а все объекты, отличные от `false` и `nil`, рассматриваются как `true`.

Листинг 11.15. Фильтрация массива при помощи итератора `select`. Файл select.rb

```
p [1, 2, 3, 4, 5].select { |x| x.even? } # [2, 4]
```

Для определения четности числа используется метод `even?`, который возвращает `true`, если число является четным, и `false` — если число нечетное:

```
 > 3.even?
=> false
 > 4.even?
=> true
```

Для того чтобы правильно составить итератор `select`, его — в целях отладки — можно заменить на итератор `map`. В результате будет получена логическая маска, содержащая элементы `true` и `false` (листинг 11.16). Так разобраться с итератором `select` гораздо проще.

Листинг 11.16. Отладка итератора `select`. Файл select_debug.rb

```
arr = [1, 2, 3, 4, 5]

p arr                     # [1, 2, 3, 4, 5]
p arr.map { |x| x.even? } # [false, true, false, true, false]
```

Программа выводит два массива: оригинальный и массив с логическими значениями. Итератор `select` извлекает из оригинальной коллекции только те элементы, для которых соответствующее значение блока возвращает значение `true` («истина»). Все элементы, для которых блок вернул `false`, — отбрасываются.

Итератор `reject` противоположен итератору `select` — в листинге 11.17 он отбирает только те элементы, для которых блок вернул значение `false` («ложь»).

Листинг 11.17. Фильтрация массива при помощи итератора `reject`. Файл reject.rb

```
arr = [1, 2, 3, 4, 5]

p arr                       # [1, 2, 3, 4, 5]
p arr.map { |x| x.even? }   # [false, true, false, true, false]
p arr.reject { |x| x.even? } # [1, 3, 5]
```

Итераторы можно объединять в цепочки. Создадим коллекцию из четных элементов, значения которых возводятся в квадрат. Отобрать четные элементы коллекции нам поможет итератор `select`, а возвести в квадрат элементы полученного массива можно при помощи итератора `map` (рис. 11.3).

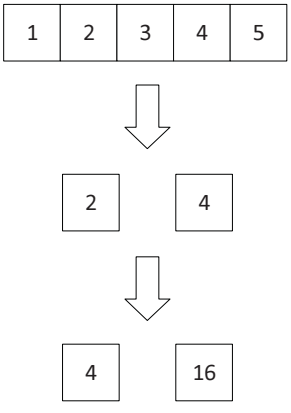


Рис. 11.3. Каскадное преобразование массива при помощи итераторов `select` и `map`

Для реализации задуманного можно вызывать один итератор за другим, не сохраняя промежуточные результаты во временных переменных (листинг 11.18).

Листинг 11.18. Итераторы можно объединять в цепочку вызовов. Файл chain.rb

```
p [1, 2, 3, 4, 5].select { |x| x.even? }.map { |x| x * x } # [4, 16]
```

11.5.5. Итератор *reduce*

Еще один итератор для работы с коллекциями — `reduce`. Его задача состоит в обходе элементов коллекции и агрегации их либо новую коллекцию, либо вообще в одно-единственное значение. Для `reduce` предусмотрен синоним: `inject`.

В листинге 11.19 приводится пример использования итератора `reduce` для умножения элементов массива друг на друга.

Листинг 11.19. Использование итератора `reduce`. Файл reduce.rb

```
p [1, 2, 3, 4, 5].reduce { |m, x| m * x } # 120
```

Если проверить результат в интерактивном Ruby (`irb`), можно убедиться, что полученный результат: `120` — верен:

```
 > 1 * 2 * 3 * 4 * 5
=> 120
```

Блок итератора `reduce` принимает два параметра: первый параметр — это объект, в котором накапливаются значения, второй — элемент коллекции на текущей итерации. В блоке оба параметра перемножаются, и получается результат.

Итератор `reduce` — это яркое проявление соглашений языка Ruby. Эффективно использовать его весьма трудно, если не знать, как он устроен. На каждой итерации переменной `m` присваивается результат блока (рис. 11.4). После того как все элементы массива обработаны, итератор возвращает содержимое переменной `m`.

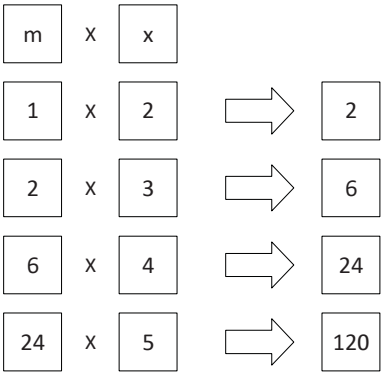


Рис. 11.4. Состояние переменных `m` и `x` в блоке

Переменная `m` инициализируется первым элементом коллекции, в приведенном примере — это цифра `1`. В большинстве случаев это вполне подходящий выбор, однако можно установить свое собственное начальное значение `m`, если передать итератору `reduce` необязательный аргумент. В листинге 10.20 в качестве начального значения устанавливается число `10`.

Листинг 11.20. Инициализация итератора `reduce`. Файл reduce_init.rb

```
p [1, 2, 3, 4, 5].reduce(10) { |m, x| m * x } # 1200
```

Допускается сокращенная форма итератора `reduce`. В этом случае в качестве аргумента итератору `reduce` передается символ, и не используется блок. Вычисление факториала числа пять — 5! — из листинга 11.19 можно записать более кратко (листинг 11.21).

Листинг 11.21. Краткая форма итератора `reduce`. Файл reduce_short.rb

```
p [1, 2, 3, 4, 5].reduce { |m, x| m * x } # 120
p [1, 2, 3, 4, 5].reduce(:*)              # 120
p [1, 2, 3, 4, 5].reduce(10, :*)          # 1200
```

Первые две строки листинга 11.21 полностью эквивалентны. Последняя строка программы показывает, как можно проинициализировать внутреннюю переменную-счетчик начальным значением 10.

11.5.6. Итератор *each_with_object*

Итератор `reduce` объединяет возможности итераторов `map`, `select` и `reject`. При помощи `reduce` можно решать одновременно две задачи: фильтровать коллекции и

преобразовывать ее элементы. Задачу получения квадрата четных элементов (см. листинг 11.18) можно решить при помощи единственного итератора `reduce`. Для этого в качестве инициализирующего значения итератору следует передать массив (листинг 11.22). В этот массив отбираются элементы по условию, заданному в `if`-модификаторе.

Листинг 11.22. Фильтрация и преобразование элементов. Файл reduce_filter.rb

```
arr = [1, 2, 3, 4, 5].reduce([]) do |m, x|
  m << x * x if x.even?
  m
end

p arr # [4, 16]
```

В качестве результата получается массив, состоящий из квадрата четных элементов исходного массива. В конце блока необходимо размещать массив `m`, в который помещаются преобразованные элементы. Если этого не сделать, то во время итерации, где `if`-модификатор не срабатывает, блок вернет значение `nil`, и переменная `m` на следующей итерации будет иметь неопределенное значение, что приведет к ошибке.

Более элегантно решить эту задачу позволяет итератор `each_with_object` (листинг 11.23). Следует обратить внимание, что в нем порядок следования параметров в блоке отличается от итератора `reduce`: первый параметр — текущий элемент коллекции, второй — объект накопления результата.

Листинг 11.23. Файл each_with_object.rb

```
arr = [1, 2, 3, 4, 5].each_with_object([]) do |x, m|
  m << x * x if x.even?
end

p arr # [4, 16]
```

Итератор `each_with_object` позволяет не указывать в конце блока возвращаемое значение, в результате чего операции фильтрации и преобразования выглядят более элегантно.

11.6. Итератор *tap*

Иногда возникает необходимость в получении промежуточного результата. Один из вариантов решения этой проблемы заключается в выводе значений внутри блока (листинг 11.24).

Листинг 11.24. Вывод промежуточных значений. Файл map_reduce_debug.rb

```
result = (1..7).select { |x| x.even? }.reduce do |m, x|
           puts "debug: #{x}"
           m + x
         end

p result
```

Результатом выполнения программы будут следующие строки:

```
debug: 4
debug: 6
12
```

Однако в этой ситуации элегантнее использовать итератор `tap` (листинг 11.25). Он возвращает исходный объект без изменений и используется только для побочного эффекта.

Листинг 11.25. Использование итератора `tap`. Файл tap.rb

```
result = (1..7).select { |x| x.even? }
               .tap { |x| puts "debug: #{x}" }
               .reduce { |m, x| m + x }

p result
```

У итератора `tap` есть и другие применения. В листинге 11.26 приводится пример, где метод `hash_return` получает в качестве параметра хэш `params`, в который вносятся изменения. В последней строке необходимо разместить хэш, чтобы метод вернул его.

Листинг 11.26. Файл hash_return.rb

```
def hash_return(params)
  params[:page] = 1
  params
end

p hash_return(per_page: 10) # {:per_page=>10, :page=>1}
```

Если не вернуть в конце хэш `params`, то в качестве результата работы метода будет возвращаться единица — значение, которое возвращается оператором присваивания:

```
def hash_return(params)
  params[:page] = 1
end

p hash_return(per_page: 10)
```

В описанной ситуации так же прибегают к итератору `tap` (листинг 11.27).

Листинг 11.27. Файл tap_return.rb

```
def hash_return(params)
  params.tap { |p| p[:page] = 1 }
end

p hash_return(per_page: 10) # {:per_page=>10, :page=>1}
```

11.7. Сокращенная форма итераторов

Если блок принимает лишь один параметр, и его работа сводится к вызову единственного метода, можно заменить блок аргументом-символом, перед которым указывается символ амперсанда. Так, выражения в листинге 11.28 являются полностью эквивалентными.

Листинг 11.28. Сокращенная форма итератора `select`. Файл select_short.rb

```
p [1, 2, 3, 4, 5].select { |x| x.even? } # [2, 4]
p [1, 2, 3, 4, 5].select(&:even?)        # [2, 4]
```

В отличие от итератора `reduce`, в котором используется обычный символ без предварительного амперсанда, форма с амперсандом может применяться для всех блоков без исключения (более подробно блоки и их свойства рассматриваются в *главе 12*).

11.8. Досрочное прекращение итерации

Внутри итераторов можно использовать те же ключевые слова, что и внутри циклов:

- `break` — прерывает работу итератора;
- `next` — прерывает текущую итерацию;
- `redo` — повторяет текущую итерацию.

В листинге 11.29 при помощи итератора `each` производится обход массива с элементами от 1 до 10. С помощью ключевого слова `break` работа итератора прерывается по достижении значения `5`.

Листинг 11.29. Использование ключевого слова `break`. Файл break.rb

```
[*1..10].each do |i|
  break if i > 5
  puts i
end
```

Результатом выполнения программы будут строки с числами от 1 до 5:

```
1
2
3
4
5
```

Ключевое слово next позволяет прервать текущую итерацию и начать следующую. В листинге 11.30 при помощи next фильтруются значения массива: выводятся только нечетные элементы.

Листинг 11.30. Использование ключевого слова next. Файл next.rb

```
[*1..10].each do |i|
  next if i.even?
  puts i
end
```

Здесь, если текущий элемент массива оказывается четным — вызывается ключевое слово next, и вывод puts не срабатывает.

Ключевое слово redo позволяет перезапустить текущую итерацию. В листинге 11.31 каждый элемент массива выводится три раза. Достигается это повторным выполнением итерации, пока переменная-счетчик j не получит значение больше трех.

Листинг 11.31. Использование ключевого слова redo. Файл redo.rb

```
[1, 2, 3, 4, 5].each do |x|
  puts x
  redo if (j ||= 0) && (j += 1) && j <= 3
end
```

Здесь на каждой итерации переменная j инициализируется нулевым значением, после чего ее значение увеличивается на единицу. Условие if-модификатора эквивалентно следующему логическому выражению:

```
true && true && j <= 3
```

Или после упрощения:

```
j <= 3
```

Ключевое слово redo срабатывает до тех пор, пока переменная j не достигнет значения 4. После этого начинается новая итерация, на которой переменная j снова имеет неопределенное значение.

11.9. Класс *Enumerator*

Допускается использование итераторов без передачи им блока — в этом случае возвращается объект класса `Enumerator` (листинг 11.32).

Листинг 11.32. Использование итераторов без блоков. Файл enumerator.rb

```
e = [*1..3].each
p e.class # Enumerator
p e.size  # 3
p e.next  # 1
p e.next  # 2
p e.next  # 3
p e.next  # iteration reached an end (StopIteration)
```

Как и любой другой объект языка Ruby, объект класса `Enumerator` поддерживает методы. Например, при помощи метода `size` можно запросить размер коллекции. Последовательный вызов метода `next` позволяет извлекать данные из коллекции. По достижении конца коллекции возникает исключительная ситуация `StopIteration`.

Ruby предоставляет итератор `each_with_index`, однако итератора `map_with_index` или `select_with_index` не предусмотрено. Похожую функциональность можно легко организовать, если воспользоваться методом `with_index` класса `Enumerator` (листинг 11.33).

Листинг 11.33. Использование метода `with_index`. Файл with_index.rb

```
rainbow = %w[red orange yellow green gray indigo violet]
arr = rainbow.map.with_index { |color, i| "#{i}: #{color}" }
p arr
```

Результатом выполнения программы станет следующий массив:

```
["0: red", "1: orange", "2: yellow", "3: green", "4: gray", "5: indigo",
                                                          "6: violet"]
```

По аналогии с итератором `each_with_object`, класс `Enumerator` предоставляет метод `with_object` (листинг 11.34).

Листинг 11.34. Использование метода `with_object`. Файл with_object.rb

```
arr = [1, 2, 3, 4, 5].map.with_object([]) do |x, m|
  m << x * x if x.even?
end
p arr # [4, 16]
```

Задания

1. Заполните массив цветов `colors` названиями цветов, полученными от пользователя методом `gets`. Количество цветов не должно быть ограничено, после ввода цвета пользователю должно быть предложено ввести дополнительный цвет. Для того чтобы прервать ввод, необходимо ввести стоп-слово `'stop'`. После этого необходимо вывести список цветов в стандартный поток вывода.
2. Используя массив, полученный в предыдущем задании, выведите цвета в алфавитном порядке. Кроме того, уберите из списка повторяющиеся цвета и пустые строки.
3. Дан массив `%w[cat dog tiger]`. Нужно вернуть массив, состоящий только из элементов, где встречается символ `t`.
4. Дан массив `%w[cat dog tiger]`. Необходимо изменить его так, чтобы все элементы были написаны с заглавной буквы.
5. Строку `'Hello world!'` преобразуйте в массив `["H", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d", "!"]`. Выражение преобразования должно быть однострочным.
6. Создайте массив с днями недели на русском языке, извлеките дни недели, которые начинаются на букву `'с'`.
7. Создайте массив с названиями месяцев на русском языке. Извлеките месяцы с самым коротким и с самым длинным названием.
8. Создайте класс пользователей `User`, который позволяет сохранять в объекте фамилию, имя, отчество пользователя, а также его оценку по предмету (в диапазоне от 1 до 5). Создайте программу, которая формирует массив 10 объектов класса `User`. Вычислите среднюю оценку и выведите список пользователей, чья оценка выше среднего.
9. Создайте хэш, в котором в качестве ключа выступает название планеты Солнечной системы, а в качестве значения — масса планеты. Сформируйте две коллекции с тремя самыми легкими и тремя самыми тяжелыми планетами. Выведите их в стандартный поток вывода.

ГЛАВА 12

Блоки

Файлы с исходными кодами этой главы находятся в каталоге *blocks* сопровождающего книгу электронного архива.

Блоки — уникальные синтаксические конструкции языка Ruby, предназначенные для группировки Ruby-выражений и отложенного их выполнения. Эти конструкции довольно интенсивно используются на практике. В предыдущей главе мы рассматривали работу блоков совместно со стандартными итераторами. Тем не менее блоки можно создавать и самостоятельно при помощи ключевого слова `yield` и объектов класса `Proc`.

Блоки очень похожи на методы тем, что выполняют Ruby-выражения и могут принимать параметры. Однако между методами и блоками существует принципиальная разница. Блоки могут хранить состояние, более того, блоки могут существовать отдельно от остальных конструкций языка в виде `Proc`-объектов. Методы же в Ruby без обращения к возможностям объектов хранить состояние не способны.

12.1. Блоки в собственных методах

Одним из самых простых итераторов является итератор `loop`, который бесконечно повторяет содержимое блока (листинг 12.1). В нашем случае он выводит фразу `'Hello, world!'`.

Листинг 12.1. Использование блока совместно с итератором `loop`. Файл loop.rb

```
loop { puts 'Hello, world!' }
```

Для того чтобы лучше разобраться в работе блоков, разработаем похожий итератор с названием `my_loop`. Итераторы — это методы. Для того чтобы метод мог принимать блок, в нем необходимо воспользоваться ключевым словом `yield` (листинг 12.2).

Листинг 12.2. Использование ключевого слова `yield`. Файл yield.rb

```
def my_loop
  puts 'Начало метода'
  yield
  puts 'Завершение метода'
end
my_loop { puts 'Hello, world!' }
```

Результатом выполнения программы из листинга 12.2 будут следующие строки:

```
Начало метода
Hello, world!
Завершение метода
```

При вызове метода `my_loop` выражения в теле метода выполняются от первого до последнего. Сначала выводится фраза `'Начало метода'`. Затем вызывается ключевое слово `yield`, которое уступает вычисление внешнему блоку. Выполняются все Ruby-выражения в блоке, и управление передается следующему за `yield` выражению. Выводится фраза `'Завершение метода'`, и метод завершает свою работу.

Блок можно вычислять многократно — для этого ключевое слово `yield` необходимо вызывать столько раз, сколько раз требуется вызвать блок (листинг 12.3).

Листинг 12.3. Трехкратное выполнение содержимого блока. Файл yield_many.rb

```
def my_loop
  puts 'Начало метода'
  yield
  yield
  yield
  puts 'Завершение метода'
end

my_loop { puts 'Hello, world!' }
```

Результатом выполнения программы будут следующие строки:

```
Начало метода
Hello, world!
Hello, world!
Hello, world!
Завершение метода
```

Метод `my_loop` здесь три раза уступал вычисления внешнему блоку. Таким образом, чтобы получить итератор, который бесконечно выполняет блок, необходимо зациклить вызов `yield`.

В листинге 12.4 циклу `while` в качестве аргумента передается «истина» — `true`. Благодаря этому цикл будет выполняться бесконечно, уступая вычисления блоку за счет вызова в теле цикла ключевого слова `yield`.

Листинг 12.4. Реализация итератора `my_loop`. Файл my_loop.rb

```
def my_loop
  while true
    yield
  end
end

my_loop { puts 'Hello, world!' }
```

Метод `my_loop` можно записать еще короче, если воспользоваться модифицированной формой цикла `while` (листинг 12.5).

Листинг 12.5. Короткая реализация итератора `my_loop`. Файл my_loop_short.rb

```
def my_loop
  yield while true
end
...
```

12.2. Передача значений в блок

После ключевого слова `yield` можно указывать произвольное количество аргументов. Аргументы подставляются в соответствующие параметры блока. Например, для передачи в блок значения `5` можно использовать вызов:

```
yield 5
```

В этом случае внутри блока следует указать параметр между двумя вертикальными чертами — например:

```
my_loop { |i| ... }
```

Значение `5` будет передано параметру `i`, который можно задействовать внутри блока. В листинге 12.6 в методе `my_loop` заводится локальная переменная `n`, значение которой увеличивается на единицу с каждым новым вызовом ключевого слова `yield`.

Листинг 12.6. Передача значений в блок. Файл block_param.rb

```
def my_loop
  n = 0
  yield n += 1
  yield n += 1
  yield n += 1
end

my_loop { |i| puts "#{i}: Hello, world!" }
```

Значение из ключевого слова `yield` подставляется в параметр `i` блока, который в свою очередь интерполируется в строку. Результатом выполнения программы будут следующие строки:

```
1: Hello, world!
2: Hello, world!
3: Hello, world!
```

В ключевое слово `yield` можно передать сразу несколько значений. В листинге 12.7 внутрь блока передаются две строки: `'Hello'` и `'Ruby'`. Эти строки попадают в параметры `interjection` и `noun`, из которых формируется строка `'Hello, Ruby!'`.

Листинг 12.7. Блок с несколькими параметрами. Файл block_params.rb

```
def greeting
  yield 'Hello', 'Ruby'
end

greeting do |interjection, noun|
  puts "#{interjection}, #{noun}!" # Hello, Ruby!
end
```

Хотя ключевое слово `yield` не накладывает ограничений на количество параметров, на практике их задействуется максимум три. В случае, когда необходимо передать больше информации, прибегают к использованию массивов, хэшей или объектов.

Если аргументов в `yield` больше, чем параметров в блоке, лишние аргументы отбрасываются. Если аргументов в `yield` меньше, чем параметров в блоке, параметры, которым не хватило аргумента, получат неопределенное значение `nil` (листинг 12.8).

Листинг 12.8. Файл block_params_diff.rb

```
def greeting
  yield 'Hello', 'Ruby', '!'
end

greeting do |interjection, noun|
  puts "#{interjection}, #{noun}!" # Hello, Ruby!
end

greeting do |fst, snd, thd, fth|
  p fst # "Hello"
  p snd # "Ruby"
  p thd # "!"
  p fth # nil
end
```

12.3. Метод *block_given?*

Если внутри метода вызывается ключевое слово `yield` — блок обязателен. При попытке вызвать такой метод без блока возникает ошибка (листинг 12.9).

Листинг 12.9. Если забыть указать блок — возникает ошибка. Файл block_error.rb

```
def my_loop
  yield while true
end

my_loop # `my_loop': no block given (yield) (LocalJumpError)
```

Впрочем, это ограничение можно обойти при помощи логического метода `block_given?`, который проверяет, передан ли методу блок (листинг 12.10).

Листинг 12.10. Использование метода `block_given?`. Файл block_given.rb

```
def my_loop
  if block_given?
    yield while true
  end
end

my_loop
```

Метод `block_given?` возвращает «истину» `true`, если текущему методу передан блок, и «ложь» `false`, если блок не передан. Таким образом цикл `while` запускается только в том случае, если метод получает блок.

Программу из листинга 12.10 можно подвергнуть рефакторингу — сразу вернуть управление из метода при помощи ключевого слова `return`, если методу не передан блок (листинг 12.11).

Листинг 12.11. Файл guard_condition.rb

```
def my_loop
  return unless block_given?
  yield while true
end
...
```

12.4. Возврат значений из блока

Блоки тоже возвращают значение, которое можно получить как результат вызова `yield`. Пока мы еще к такому возвращаемому значению не обращались. Однако этот подход является мощным средством создания обобщенных библиотек.

Методы, классы и модули часто создаются как инструменты, которые будут использовать другие разработчики. Причем зачастую неизвестно, в каких условиях будет работать класс: в составе консольной утилиты, приложения с графическим интерфейсом или на сервере.

При помощи блоков можно оставить возможность адаптировать приложение под задачи внешнего разработчика. В листинге 12.12 приводится пример метода `greeting`, который возвращает фразу `'Hello, world!'`. При этом слово `world` может быть заменено содержимым из блока.

Листинг 12.12. Возврат значения из блока. Файл block_return.rb

```
def greeting
  name = block_given? ? yield : 'world'
  "Hello, #{name}!"
end

puts greeting            # Hello, world!
puts greeting { 'Ruby' } # Hello, Ruby!

hello = greeting do
  print 'Пожалуйста, введите имя '
  gets.chomp
end
puts hello
```

В программе метод `greeting` вызывается три раза: без блока, с передачей из блока внутрь метода строки `'Ruby'` и с блоком, в котором имя запрашивается у пользователя при помощи метода `gets`. Результат выполнения программы может выглядеть следующим образом:

```
$ ruby block_return.rb
Hello, world!
Hello, Ruby!
Пожалуйста, введите имя Igor
Hello, Igor!
```

12.5. Итератор *yield_self*

В *главе 11* рассматривался итератор `tap`, который всегда возвращает оригинальное значение. Итератор `yield_self` — наоборот, всегда возвращает результат вычисления блока. Оба итератора схожи тем, что могут быть применены к любому объекту языка Ruby:

```
 > 5.tap { |x| x * x }
=> 5
 > 5.yield_self { |x| x * x }
=> 25
```

В листинге 12.13 приводится использование yield_self для формирования строки приветствия, аналогичной той, что была сформирована в предыдущем разделе.

Замечание

Метод yield_self добавлен в Ruby, начиная с версии 2.5.3.

Листинг 12.13. Возврат значения из блока. Файл yield_self.rb

```
hello = 'Hello, %s!'.yield_self do |template|
          print 'Пожалуйста, введите имя '
          format(template, gets.chomp)
        end
puts hello
```

При помощи итератора yield_self внутрь блока передается содержимое строки 'Hello, %s!'. Внутри блока строка передается в переменную template. При помощи метода gets у пользователя запрашивается имя, которое подставляется в шаблон template при помощи метода format.

Результат выполнения программы может выглядеть следующим образом:

```
$ ruby yield_self.rb
Пожалуйста, введите имя Igor
Hello, Igor!
```

Итератору yield_self не обязательно использовать исходный объект — мы можем полностью заменить его новым:

```
 > 'Ruby'.yield_self { |_x| 'Hello, world!' }
=> "Hello, world!"
```

В приведенном примере параметр блока вообще не задействуется для формирования возвращаемого значения, поэтому он предваряется символом подчеркивания. Это сообщает читающему код, что переменная не используется. Впрочем, в случае yield_self, параметр в блоке можно вообще убрать:

```
 > 'Ruby'.yield_self { 'Hello, world!' }
=> "Hello, world!"
```

12.6. Передача блока через параметр

Блок можно оформить в виде специального параметра метода — более того, его можно указать явно. Для этого перед последним параметром метода указывается символ амперсанда &. Такой параметр в методе может быть только один, и он должен располагаться последним.

В листинге 12.14 в качестве такого параметра выступает block. Название параметру можно назначать произвольное.

Листинг 12.14. Передача блока через параметр. Файл block_proc.rb

```
def greeting(&block)
  block.call
end

greeting { puts 'Hello, world!' }
```

В случае явной передачи блока, вместо вызова ключевого слова `yield` у объекта `block` вызывается метод `call`. Вызов этого метода уступает вычисление блоку.

Как и в случае ключевого слова `yield`, метод `call` может принимать произвольное количество аргументов, которые становятся параметрами блока. Метод `call` возвращает результат вычисления блока (листинг 12.15).

Листинг 12.15. Передача блока через параметр. Файл call.rb

```
def greeting(&block)
  block.call 'Ruby!'
end

puts greeting { |name| "Hello, #{name}!" } # Hello, Ruby!
```

В листинге 12.15 блок возвращает строку `'Hello, Ruby!'`, которую сначала возвращает вызов `block.call` в методе `greeting`. Поскольку это последнее выражение метода, метод `greeting` так же возвращает строку `'Hello, Ruby!'`. Результат метода `greeting` передается в качестве аргумента методу `puts`.

Зачем вообще потребовалась альтернатива ключевому слову `yield`? При использовании `yield` блок обязательно должен быть передан тому методу, в котором `yield` вызван. Это может быть неудобно, если методы вызывают друг друга (листинг 12.16).

Листинг 12.16. Нельзя вызывать блок у метода `outer`. Файл nested_methods.rb

```
def greeting(name)
  yield name
end

def outer(name)
  greeting(name) { |name| "Hello, #{name}!" }
end

puts outer('Ruby') # Hello, Ruby!
```

Метод `outer` вызывает метод `greeting`, в котором происходит вызов ключевого слова `yield`. При этом блок приходится передать методу `greeting`, поскольку нет возможности вызвать его на уровне выше — у метода `outer` (рис. 12.1).

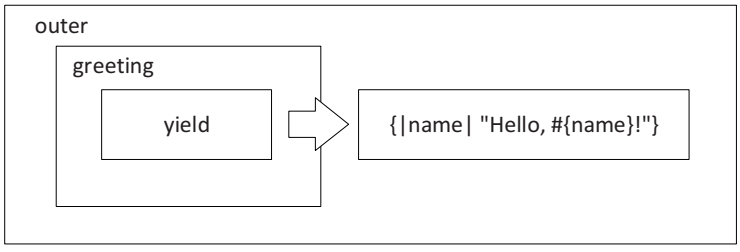


Рис. 12.1. Блок передается методу, в котором вызывается ключевое слово `yield`

В случае передачи блока через параметр, блок можно вызывать в отношении любого метода, который принимает параметр с `&` (листинг 12.17).

Листинг 12.17. Файл nested_methods_proc.rb

```
def greeting(name, &block)
  block.call name
end

def outer(name, &block)
  greeting(name, &block)
end

str = outer('Ruby') { |name| "Hello, #{name}!" }
puts str # Hello, Ruby!
```

Благодаря тому, что блок передается через параметр, — его можно передавать пользователю с более глубокого уровня вызова. Например, можно вызывать метод `call` во вложенном методе `greeting`. За счет использования параметра `&block` блок можно применить к методу `outer`. То есть параметрами можно передать блок как угодно далеко от точки вызова метода `call` (рис. 12.2).

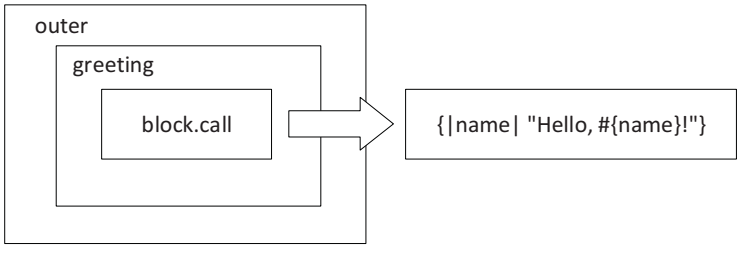


Рис. 12.2. Параметры позволяют передать блок далеко от точки применения блока

12.7. Различие *{ ... }* и *do ... end*

В *главе 11* мы рассмотрели две формы блоков: с использованием фигурных скобок `{ ... }` и ключевых слов `do ... end` (листинг 12.18).

Листинг 12.18. Формы блоков. Файл block_form.rb

```
loop { puts 'Hello, world!' }
loop do
  puts 'Hello, world!'
end
```

До сих пор мы считали их взаимозаменяемыми. Однако между этими формами существует тонкое отличие: фигурные скобки обладают более высоким приоритетом по сравнению с блоком `do ... end`. В листинге 12.19 приводится пример преобразования массива при помощи итератора `map`. Все элементы массива возводятся в квадрат. Действие выполняется два раза с разными формами блоков.

Листинг 12.19. Некорректное использование блока. Файл map.rb

```
p [1, 2, 3, 4, 5].map { |x| x * x }
p [1, 2, 3, 4, 5].map do |x|
    x * x
  end
```

Результатом выполнения программы будут следующие строки:

```
[1, 4, 9, 16, 25]
#<Enumerator: [1, 2, 3, 4, 5]:map>
```

Вариант с фигурными скобками выдал ожидаемый результат, в то время как в случае полного блока был возвращен неожиданный вариант.

Блок — это параметр метода. В формировании результата участвуют два метода: `p` и `map`. Круглые скобки вокруг аргументов необязательны и на практике часто опускаются. Соблазн их опустить еще больше возрастает, когда метод принимает блок.

Как уже было отмечено, приоритет фигурных скобок `{ ... }` и ключевых слов `do ... end` различен. То есть программа из листинга 12.19 эквивалентна варианту из листинга 12.20.

Листинг 12.20. Некорректное использование блока. Файл map_bracket.rb

```
p( [1, 2, 3, 4, 5].map { |x| x * x } )
p( [1, 2, 3, 4, 5].map ) do |x|
                           x * x
                         end
```

В первом случае блок получает метод `map`, и после выполнения преобразования массива метод `p` получает массив `[1, 4, 9, 16, 25]`, который выводится в качестве результата.

Во втором случае метод `p` получает в качестве первого параметра объект класса `Enumerator`:

```
 > [1, 2, 3, 4, 5].map
=> #<Enumerator: [1, 2, 3, 4, 5]:map>
```

а в качестве второго параметра метод p получает блок:

```
p([1, 2, 3, 4, 5].map, &block)
```

Блок вполне допустим в случае метода p, правда, его передача не приводит к каким-либо результатам:

```
> p { |x| x * x}
 => nil
> p do |x| x * x end
 => nil
```

Для того чтобы получить массив [1, 4, 9, 16, 25] в случае блока do ... end, необходимо воспользоваться либо круглыми скобками, либо вынести результат в отдельную локальную переменную (листинг 12.21).

Листинг 12.21. Файл map_full_block.rb

```
p( [1, 2, 3, 4, 5].map do |x| x * x end )

arr = [1, 2, 3, 4, 5].map do |x|
        x * x
      end
p arr
```

12.8. Блоки в рекурсивных методах

Использование ключевого слова yield — самый простой и наглядный способ организации блоков. Однако в ряде случаев передача блоков через параметр — единственный способ воспользоваться блоком. К ним относятся рекурсивные методы, т. е. методы, которые вызывают сами себя (см. *разд. 9.7*).

Рекурсивные методы особенно полезны при обходе древовидных структур. Пусть имеется файловая система со множеством вложенных каталогов и файлов. Зададимся целью создать итератор, который предоставляет список файлов в виде линейного списка.

Обычно эта задача решается в рамках специальных классов Dir и File (см. *главы 27–29*). Чтобы не забегать вперед, представим, что структура каталогов и файлов оформлена в виде массива. Каталоги в этом массиве представлены в виде хэшей, а файлы — в виде строк (листинг 12.22).

Листинг 12.22. Древовидная структура. Файл tree.rb

```
TREE = [
  'index.rb',
  {
    'src' =>  [
```

```
      'file01.rb',
      'file02.rb',
      'file03.rb',
    ]
  },
  {
    'doc' => [
      'file01.md',
      'file02.md',
      'file03.md',
      {
        'details' => [
          'index.md',
          'arch.md'
        ]
      }
    ]
  }
]
```

Массив специально размещается в константе `TREE`, чтобы его можно было получить в другом Ruby-файле. Для этого файл tree.rb, в котором размещается массив, нужно подключить при помощи методов `require` и `require_relative`. Локальную переменную передать таким способом не получится.

После подключения файла можно воспользоваться массивом `TREE` (листинг 12.23).

Листинг 12.23. Обход массива. Файл tree_each.rb

```
require_relative 'tree'
TREE.each { |x| p x }
```

Если воспользоваться готовым итератором `each`, будет затронут только верхний уровень древовидной структуры (элементы массива `TREE`). В результате выводятся три строки:

```
"index.rb"
{"src"=>["file01.rb", "file02.rb", "file03.rb"]}
{"doc"=>["file01.md", "file02.md", "file03.md", {"details"=>["index.md",
                                                    "arch.md"]}]}
```

Первая строка — файл `'index.rb'`, две другие — каталоги `'src'` и `'doc'`. Для начала научим Ruby-программу отличать каталог от файла (листинг 12.24).

Листинг 12.24. Обход массива. Файл tree_file_or_dir.rb

```
require_relative 'tree'

TREE.each do |x|
  puts 'Dir' if Hash === x
```

```
  puts 'File' if String === x
end
```

Оператор === перегружен в классах таким образом, чтобы при сравнениях с объектами возвращать «истину» true, если объект принадлежит классу, и «ложь» false — в противном случае. Таким образом, если объект является хэшем, считаем его каталогом, если строкой — файлом. Результатом работы программы будут следующие строки:

```
File
Dir
Dir
```

Можно переписать программу из листинга 12.24 с использованием ключевого слова case, т. к. оно неявно использует оператор === (листинг 12.25).

Листинг 12.25. Обход массива. Файл tree_case.rb

```
require_relative 'tree'

TREE.each do |x|
  case x
  when Hash then puts 'Dir'
  when String then puts 'File'
  end
end
```

Такой способ определения класса объекта является в Ruby-программах весьма распространенной практикой. Впрочем, как будет показано в следующих главах, лучше не ориентироваться на класс объекта. При желании в Ruby любой объект можно превратить в строку, массив или хэш, а поведение любого, даже предопределенного, класса изменить до неузнаваемости.

Теперь все готово для создания собственного итератора. Для этого код обхода размещаем в методе walk (листинг 12.26).

> ***Замечание***
>
> Обратите внимание, что в случае сложной программы ее разработка ведется методом последовательного приближения. В реальности разработчики не пишут сразу много кода, отчаянно отлавливая в нем потом ошибки, а двигаются небольшими перебежками от одной работающей версии до другой.

Листинг 12.26. Обход массива. Файл tree_method.rb

```
require_relative 'tree'

def walk(tree = [])
  tree.each do |x|
    case x
```

```
    when Hash then puts 'Dir'
    when String then puts 'File'
    end
  end
end

walk(TREE)
```

Массив `TREE` передается в качестве параметра методу `walk`. При обходе коллекции `tree` необходимо для каждого каталога повторно вызывать метод `walk`, т. к. внутри каталога могут быть свои собственные файлы (листинг 12.27).

Листинг 12.27. Рекурсивный обход коллекции. Файл tree_recursion.rb

```
require_relative 'tree'

def walk(tree = [])
  tree.each do |x|
    case x
    when Hash
      x.each { |_dir, files| walk(files) }
    when String then puts x
    end
  end
end

walk(TREE)
```

Вместо сообщения о том, что в массиве встретился каталог, метод `walk` вызывается в отношении этого каталога. Таким образом, метод `walk` превращается в рекурсивный, поскольку вызывает сам себя. Вместо сообщения о том, что элемент является файлом, просто выводим его название при помощи метода `puts`. Результатом работы программы из листинга 12.27 будет следующий список файлов:

```
index.rb
file01.rb
file02.rb
file03.rb
file01.md
file02.md
file03.md
index.md
arch.md
```

Наша исходная цель заключалась в создании итератора, т. е. метода, который может принимать блок. Так как в случае рекурсии невозможно предсказать уровень вложенности вызовов метода `walk`, для организации блока не получится воспользоваться ключевым словом `yield`. В этом случае пригодится передача блоков через параметр (листинг 12.28).

Листинг 12.28. Рекурсивный обход коллекции. Файл tree_walk.rb

```
require_relative 'tree'

def walk(tree = [], &block)
  tree.each do |x|
    case x
    when Hash
      x.each { |_dir, files| walk(files, &block) }
    when String
      block.call(x)
    end
  end
end

walk(TREE) { |file| puts file }
```

Методу `walk` передается второй параметр `&block`, который явно указывается при рекурсивном вызове. В случае файла его имя передается во внешний блок путем передачи его в качестве аргумента методу `call`. Результат работы точно такой же, как и в случае программы из листинга 12.27.

При помощи собственного итератора были отброшены все каталоги, а сложная древовидная структура вытянута в линейный список.

12.9. Класс *Proc*

Ruby предоставляет специальный класс `Proc`, объекты которого могут выступать в качестве блоков. Передача блока через параметр метода осуществляется через объект класса `Proc`.

Сам по себе блок не является объектом, однако `proc`-объект может выступать в качестве блока. Создать блок можно при помощи метода `new` класса `Proc`. В листинге 12.29 `proc`-объект используется в итераторе `select` вместо блока.

Листинг 12.29. Создание объекта класса `Proc`. Файл proc.rb

```
block = Proc.new { |x| x.even? }
p [*1..10].select(&block) # [2, 4, 6, 8, 10]
```

Метод `new` — не единственный способ получить `proc`-объект. Один из распространенных способов — применение метода `to_proc` к символу:

```
 > :even?.to_proc
=> #<Proc:0x00007f961a8876d0(&:even?)>
```

Полученный объект может использоваться вместо блока из листинга 12.29. Более того, при использовании амперсанда совместно с символами нет необходимости явно использовать метод `to_proc` (листинг 12.30).

Листинг 12.30. Создание блока `&:even?`. Файл proc_short.rb

```
p [*1..10].select(&:even?) # [2, 4, 6, 8, 10]
```

Когда Ruby-интерпретатор встречает выражение `&:even?`, происходит неявный вызов метода `to_proc` в отношении символа `:even?`.

Следует обратить внимание, что помимо `proc`-объекта `&:*` итератор `reduce` поддерживает собственную форму сокращения — символ без амперсанда `:*`. В листинге 12.31 все три формы являются эквивалентными. Для других итераторов амперсанд перед символом обязателен.

Листинг 12.31. Сокращение итератора `reduce`. Файл reduce.rb

```
p [1, 2, 3, 4, 5].reduce { |m, x| m * x } # 120
p [1, 2, 3, 4, 5].reduce(&:*)             # 120
p [1, 2, 3, 4, 5].reduce(:*)              # 120
```

В объектах класса `Symbol` метод `to_proc` реализован по умолчанию, однако никто не мешает реализовать метод `to_proc` в любом объекте Ruby и использовать его в качестве параметра итератора. В листинге 12.32 создается объект `obj`, для которого реализуется синглетон-метод `to_proc`. Такой объект можно использовать в качестве `proc`-объекта с любым методом, ожидающим блок.

Листинг 12.32. Объект, реализующий метод `to_proc`. Файл to_proc.rb

```
obj = Object.new

def obj.to_proc
  Proc.new { |x| x * x }
end

p (1..5).map(&obj) # [1, 4, 9, 16, 25]
```

12.10. Методы *proc* и *lambda*

На практике для создания объекта метод `new` класса `Proc` используют весьма редко. Чаще прибегают к методам `proc` и `lambda`:

```
 > pr = proc { |n| n * n }
=> #<Proc:0x00007f987102c298@(irb):1>
 > pr.call(3)
=> 9
 > lb = lambda { |n| n * n }
=> #<Proc:0x00007f987108f438@(irb):3 (lambda)>
 > lb.call(3)
=> 9
```

Объекты pr и lb, полученные при помощи методов proc и lambda, могут использоваться в качестве параметров блока (листинг 12.33).

Листинг 12.33. Использование метода proc. Файл proc_block.rb

```
pr = proc { |m, n| m * n }
p (1..5).reduce(&pr) # 120
```

При желании можно самостоятельно создать аналог метода proc (листинг 12.34).

Листинг 12.34. Разработка собственного варианта метода proc. Файл my_proc.rb

```
def my_proc(&pr)
  pr
end

block = my_proc { |m, n| m * n }
p (1..5).reduce(&block) # 120
```

Методы proc и lambda очень похожи, однако они отличаются свойствами. Чтобы подчеркнуть это, вместо вызова метода lambda используется специальный синтаксический конструктор ->:

```
 > lb = ->(x) { x * x }
=> #<Proc:0x00007febd50e0768@(irb):1 (lambda)>
 > lb.call(3)
=> 9
 > (->(x) { x * x }).call(3)
=> 9
```

Параметры блока в синтаксическом конструкторе -> указываются не в блоке, а в круглых скобках перед блоком.

12.11. Различия *proc* и *lambda*

Из предыдущего раздела может сложиться впечатление, что методы proc и lambda полностью эквивалентны. Тем не менее, если метод proc — это короткая форма Proc.new, то поведение lambda по сравнению с обычным proc-объектом изменено. Более того, предусмотрен отдельный логический метод lambda?, который позволяет отличить один тип объекта от другого:

```
 > pr = proc { |n| n * n }
=> #<Proc:0x00007f87eb873f78@(irb):1>
 > pr.lambda?
=> false
 > lb = ->(x) { x * x }
=> #<Proc:0x00007f87ed82fd30@(irb):3 (lambda)>
 > lb.lambda?
=> true
```

Использование ключевого слова `return` внутри метода приводит к выходу из метода — не важно, где он расположен: за пределами или внутри блока. В листинге 12.35 создается метод `five`, который принимает массив и выводит только первые пять элементов этого массива.

Листинг 12.35. Ключевое слово `return` в `proc`-блоке. Файл return_proc.rb

```
def five(arr)
  arr.each_with_index do |el, i|
    return if i >= 5
    yield el
  end
end

colors = %i[red orange yellow green blue indigo violet]
five(colors) { |color| puts color }
```

Результатом работы программы из листинга 12.35 будут первые пять цветов радуги из массива `colors`:

```
red
orange
yellow
green
blue
```

Каждый элемент массива передается во внешний блок при помощи ключевого слова `yield`. Для выхода из метода применяется ключевое слово `return`. Его использование приводит к выходу именно из метода, а не из блока. По умолчанию `return` возвращает неопределенное значение `nil`. Если же ключевое слово `return` не выполняется из-за того, что массив содержит меньше пяти элементов, возвращается массив:

```
colors = %i[red orange yellow green blue indigo violet]
p five(colors) { |_c| } # nil
p five(1..3) { |_c| }   # 1..3
```

Использование метода `return` внутри `lambda` приводит к выходу из блока, а не из самого метода. В листинге 12.36 применяется тот же самый итератор `five`, однако вместо `proc`-блока он реализован с использованием `lambda`-блока.

Листинг 12.36. Ключевое слово `return` в `lambda`-блоке. Файл return_lambda.rb

```
def five(arr)
  lb = ->(el, i) do
    return if i >= 5
    yield el
  end
```

```
  arr.each_with_index(&lb)
end

colors = %i[red orange yellow green blue indigo violet]
five(colors) { |color| puts color }
```

Результат выполнения программы полностью эквивалентен предыдущему варианту программы (см листинг 12.35). Однако при такой реализации метод `five` всегда будет возвращать исходный массив, не зависимо от исходного размера:

```
colors = %i[red orange yellow green blue indigo violet]
p five(colors) { |_c| } # [:red, :orange, :yellow, :green, :blue, ...]
p five(1..3) { |_c| }   # 1..3
```

Различия в поведении ключевого слова `return` в `proc`- и `lambda`-блоках обобщаются на рис. 12.3.

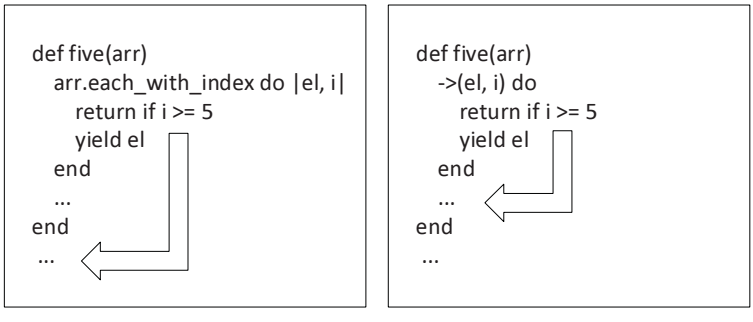


Рис. 12.3. Различие в поведении `return` внутри `proc`- и `lambda`-блоков

Еще одно различие `proc`- и `lambda`-блоков заключается в их отношении к параметрам. В `proc`-блоки можно передавать неполный набор параметров (листинг 12.37).

Листинг 12.37. Можно опускать параметры `proc`-блока. Файл proc_params.rb

```
pr = proc do |fst, snd, thd|
      p fst
      p snd
      p thd
    end

pr.call('first')
```

Результатом выполнения программы будут следующие строки:

```
"first"
nil
nil
```

В случае `lambda`-блока необходимо передавать все параметры (листинг 12.38).

Листинг 12.38. В `lambda`-блоках параметры обязательны. Файл lambda_params.rb

```
lb = lambda do |fst, snd, thd|
      p fst
      p snd
      p thd
    end

lb.call('first')
```

Выполнение этой программы завершится ошибкой, сообщающей, что количество аргументов не соответствует ожидаемому:

```
wrong number of arguments (given 1, expected 3) (ArgumentError)
```

Таким образом, `lambda`-блоки повторяют поведение методов: жесткий контроль аргументов, `return` возвращает управление из блока. В то время как `proc`-объекты повторяют поведение блоков: отсутствие контроля за количеством аргументов, `return` возвращает управление из метода.

Задания

1. Создайте полный аналог стандартного итератора `map`. Назовите его `my_map`.
2. Создайте полный аналог стандартного итератора метода `select`. Назовите его `my_select`.
3. Создайте полный аналог стандартного итератора метода `reduce`. Назовите его `my_reduce`.
4. Создайте метод `walk`, который принимает в качестве единственного аргумента массив произвольной степени вложенности:

   ```
   arr = [[[1, 2], 3], [4, 5, 6], [7, [8, 9]]]
   ```

   Метод должен реализовывать блок с единственным параметром, в который элементы вложенной последовательности передаются в виде линейного списка

   ```
   walk(arr) { |i| puts i }
   ```

   Для массива `arr` параметр `i` должен пробегать значения от 1 до 9.
5. Числа Фибоначчи — это последовательность вида 0, 1, 1, 2, 3, 5, ..., где каждое число является суммой двух предыдущих чисел. Создайте метод `fibonacci`, который принимает в качестве аргумента порядковый номер числа Фибоначчи, а в блок передает последовательность чисел от нуля до заданного числа:

   ```
   fibonacci(10) { |f| print "#{f} " } # 0 1 1 2 3 5 8 13 21 34 55
   ```
6. Создайте метод `week`, который принимает в качестве аргумента порядковый номер недели в году. В блок метода должна передаваться последовательность объектов класса `Date` с датами заданной недели: от понедельника до воскресенья.
7. Создайте метод `weekends`, который извлекает выходные дни (субботы и воскресенья) для текущего года. В блок метода должна передаваться последовательность объектов класса `Date` с выходными от начала до конца года.

ГЛАВА 13

Классы

Файлы с исходными кодами этой главы находятся в каталоге *classes* сопровождающего книгу электронного архива.

Классы позволяют создавать объекты. Кроме того, именно через класс можно задать поведение объекта, определить, какие методы и операторы допускается применять в отношении объектов.

До сих пор мы, в основном, занимались предопределенными классами, которые предоставляет язык программирования Ruby. В этой главе мы рассмотрим создание своих собственных классов, а также изучим их многочисленные свойства.

13.1. Создание класса

Для создания класса используется ключевое слово `class`, после которого указывается имя класса в CamelCase-стиле. Завершается класс ключевым словом `end`. В листинге 13.1 представлен минимально возможный класс.

Листинг 13.1. Минимально возможный класс `HelloWorld`. Файл hello_world.rb

```
class HelloWorld
end
```

Класс `HelloWorld` можно использовать в другом Ruby-файле, если подключить при помощи метода `require_relative` файл hello_world.rb. Для того чтобы создать объект этого класса, следует воспользоваться методом `new` (листинг 13.2).

Листинг 13.2. Создание объекта класса `HelloWorld`. Файл hello.rb

```
require_relative 'hello_world'

hello = HelloWorld.new
p hello # #<HelloWorld:0x007ff20606dab0>
```

Пока класс `HelloWorld` не умеет ничего, чего бы не мог стандартный класс `Object`. Однако в теле класса между ключевыми словами `class` и `end` можно размещать методы (листинг 13.3).

Листинг 13.3. Инстанс-метод `HelloWorld#say`. Файл instance_method.rb

```
class HelloWorld
  def say
    'hello'
  end
end

greeting = HelloWorld.new
hello = HelloWorld.new

puts greeting.say # hello
puts hello.say    # hello
```

Методы, созданные таким способом, можно вызывать для всех объектов классов. Поскольку эти методы вызываются для объектов, или инстансов классов, их называют *инстанс-методами*.

13.2. Класс — это объект

Класс `HelloWorld` сам является объектом, к которому мы обращаемся при помощи константы `HelloWorld` (рис. 13.1).

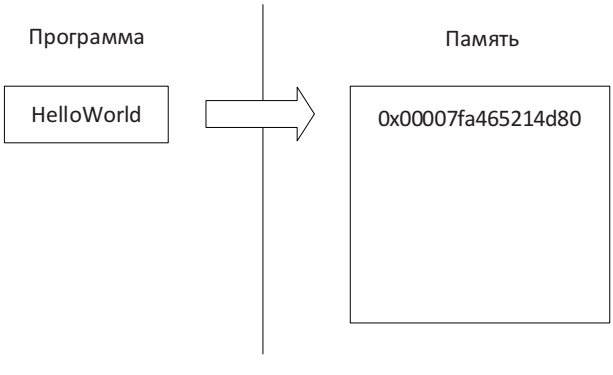


Рис. 13.1. Константа `HelloWorld` является ссылкой на объект класса

Запустим интерактивный Ruby (`irb`) и загрузим в него файл instance_method.rb:

```
 > require_relative 'instance_method'
hello
hello
=> true
```

```
 > HelloWorld
=> HelloWorld
 > HelloWorld.class
=> Class
```

Константа `HelloWorld` ссылается на объект класса `Class`. Если у объекта есть класс, можно воспользоваться методом `new` для создания объектов этого класса (листинг 13.4).

Листинг 13.4. Альтернативный способ создания класса. Файл class_new.rb

```
HelloWorld = Class.new
hello = HelloWorld.new
p hello # #<HelloWorld:0x00007ffce40e38f8>
```

Для добавления инстанс-методов в класс можно воспользоваться блоком, который передается методу `Class.new` (листинг 13.5).

Листинг 13.5. Файл class_new_block.rb

```
HelloWorld = Class.new do
  def say
    'hello'
  end
end

hello = HelloWorld.new
puts hello.say # hello
```

После ключевого слова `class` нельзя использовать имена, которые начинаются с прописной (заглавной) буквы — в этом случае возникает ошибка. Однако, если воспользоваться конструктором класса `Class`, это вполне возможно (листинг 13.6).

Листинг 13.6. Создание класса, нарушающего соглашения. Файл class_wrong.rb

```
hello_world = Class.new do
  def say
    'hello'
  end
end

hello = hello_world.new
puts hello.say # hello
```

При помощи метода `Class.new` имеется возможность создавать Ruby-классы в обход соглашений Ruby-сообщества. Однако делать этого не следует, поскольку такие классы невозможно извлечь с помощью методов `require` и `require_relative`.

Кроме того, не стоит увлекаться созданием классов через метод `Class.new` — ключевое слово `class` является более наглядным, читаемым и используется для создания 99% классов. Создание класса через `Class.new` относится к приемам метапрограммирования.

13.3. Как проектировать классы?

Объектно-ориентированное программирование (ООП) — это не самая простая область. Существует большое количество литературы, посвященной основам объектно-ориентированного программирования, хорошим практикам, типовым решениям или объектно-ориентированным паттернам программирования. Полное освещение теории ООП выходит за рамки этой книги.

Тем не менее, когда вы только начинаете осваивать объектно-ориентированное программирование, существует простое эмпирическое правило по выбору классов и методов: при проектировании системы описывается сама система и ситуации, которые могут возникать при ее использовании. В одну колонку выписываются существительные, в другую — глаголы. Существительные становятся кандидатами в классы и объекты, глаголы — в методы (рис. 13.2).

Игрок (User)	Добавить (add)
Игровое поле (SeaBattle)	Удалить (delete)
Координата (Coord)	Проверить (check)
Корабль (Ship)	Выстрелить (shoot)
Выстрел (Shoot)	Отрисовать (show)

Рис. 13.2. Декомпозиция игры «Морской бой»

Замечание

В *главах 19–21* рассмотрены модули языка Ruby, их часто удобно описывать в виде прилагательных.

Впрочем, увлекаться такими сравнениями и доводить ситуацию до абсурда не стоит — не каждое отобранное слово обязательно должно стать объектом, методом, классом или модулем. Иногда проще обойтись обычной переменной.

13.4. Переопределение методов

В классах допускается переопределять методы. В листинге 13.7 приводится пример класса, в котором метод `say` определен два раза. В Ruby это вполне допустимая ситуация — методы можно переопределять любое количество раз. Интерпретатор не будет возбуждать ошибку или выводить замечание.

Листинг 13.7. Переопределение метода. Файл method_override.rb

```
class HelloWorld
  def say
    'Определяем метод say в первый раз'
  end
  def say
    'Определяем метод say во второй раз'
  end
end

hello = HelloWorld.new
puts hello.say # Определяем метод say во второй раз
```

При анализе программы из листинга 13.7 интерпретатор Ruby будет двигаться сверху вниз. Зарегистрирует первый метод `say`, дойдет до второго и переопределит его. В результате первый метод `say` никогда не будет вызываться.

13.5. Открытие класса

В предыдущем разделе в классе `HelloWorld` метод `say` был определен два раза. Можно пойти еще дальше — Ruby позволяет повторно открывать классы и переопределять методы (листинг 13.8).

Листинг 13.8. Открытие класса. Файл class_open.rb

```
class HelloWorld
  def say
    'Определяем метод say в первый раз'
  end
end

class HelloWorld
  def say
    'Определяем метод say во второй раз'
  end
end

hello = HelloWorld.new
puts hello.say # Определяем метод say во второй раз
```

Схема работы программы остается неизменной. Интерпретатор Ruby при анализе программы движется сверху вниз. Если класс встречается впервые, он создается. Если выражение `class HelloWorld` встречается повторно, класс открывается и в него вносятся изменения. Если определение метода встречается несколько раз, будет использоваться последний определенный вариант.

Можно открывать не только свои собственные классы, но и стандартные классы Ruby. В листинге 13.9 открывается стандартный класс `String` и изменяется поведение метода `reverse`.

Листинг 13.9. Открытие стандартного класса `String`. Файл string_open.rb

```
puts 'abc'.reverse # cba

class String
  def reverse
    'Изменяем поведение метода'
  end
end

puts 'abc'.reverse # Изменяем поведение метода
```

Программа ломает стандартный класс `String`: метод `reverse` теперь ведет себя совершенно не так, как он описан в документации.

В стандартном классе `String` можно переопределять не только существующие методы, но и вводить свои собственные (листинг 13.10).

Листинг 13.10. Добавление метода `hello` всем строкам. Файл string_hello.rb

```
class String
  def hello
    "Hello, #{self}!"
  end
end

puts 'world'.hello # Hello, world!
puts 'Ruby'.hello  # Hello, Ruby!
puts 'Igor'.hello  # Hello, Igor!
```

Всем строкам добавляется метод `hello`, который возвращает измененную строку. Метод вставляет текущее значение строки в шаблон-приветствие `"Hello, #{self}!"`. Для получения текущего значения строки используется ключевое слово `self`. Оно ссылается на текущий объект и более подробно будет рассмотрено в *главе 16*.

Изменению можно подвергать не только строки, но и любые другие классы языка Ruby. Например, можно изменить поведение класса `Integer`, который определяет поведение целых чисел (листинг 13.11).

Листинг 13.11. Открытие класса `Integer`. Файл integer_open.rb

```
price = 500

puts "Цена билета #{price} рублей" # Цена билета 500 рублей
```

```
class Integer
  def to_s
    'Мы все сломали'
  end
end

puts "Цена билета #{price} рублей" # Цена билета Мы все сломали рублей
```

В приведенном примере переопределяется метод `to_s` — вместо строкового представления числа он возвращает подстроку `'Мы все сломали'`. В результате первая фраза выводится корректно, а попытка использования интерполяции после переопределения метода `Integer#to_s` приводит к искаженным результатам.

Техника открытия классов весьма опасна. Влиять можно не только на поведение своих собственных классов. Ошибки можно внести в стандартные классы Ruby и классы гемов.

Как правило, к открытию классов прибегают в том случае, когда необходимо модифицировать язык. В Ruby изначально заложена идея построения декларативных систем и фактически новых языков программирования. Например, фреймворк для разработки веб-сайтов Ruby on Rails открывает и изменяет практически каждый класс языка Ruby. Часто про Ruby on Rails говорят как о другом языке программирования, что не корректно, поскольку это обычный Ruby-код. Однако правила, поведение и соглашения Ruby on Rails весьма сильно отличаются от оригинального языка программирования Ruby.

Техника открытия класса — это инструмент разработки библиотек и новых систем. Его не следует рассматривать как повседневный инструмент создания прикладных приложений.

13.6. Тело класса и его свойства

Как было показано в *разд. 13.2*, тело класса, расположенное между ключевыми словами `class` и `end`, на самом деле является блоком. Это означает, что между этими ключевыми словами можно размещать любой Ruby-код (листинг 13.12).

Листинг 13.12. Переопределение метода. Файл class_body.rb

```
class HelloWorld
  puts 'Начало класса HelloWorld'
  puts 'Завершение класса HelloWorld'
end

hello = HelloWorld.new
p hello
```

Результатом выполнения программы будут следующие строки:

```
Начало класса HelloWorld
Завершение класса HelloWorld
#<HelloWorld:0x00007fe69217b418>
```

Более того, методы можно определять по условию. В листинге 13.13 метод `say` переопределяется в зависимости от текущей версии Ruby.

Листинг 13.13. Файл method_override_by_version.rb

```
class HelloWorld
  def say
    'Определяем метод say'
  end
  if RUBY_VERSION == '2.5.3'
    def say
      'Переопределяем метод say'
    end
  end
end

hello = HelloWorld.new
p hello.say
```

13.7. Вложенные классы

Классы можно вкладывать друг в друга. В листинге 13.14 приводится пример класса автомобиля `Car`. Так как двигатель практически не используется в отрыве от автомобиля, класс `Engine` размещается прямо внутри класса `Car`.

Листинг 13.14. Вложенные классы. Файл car.rb

```
class Car
  def title
    'BMW X7'
  end
  def description
    'Новый BMW X7 с окраской кузова...'
  end
  def engine
    @engine
  end
  def build
    @engine = Engine.new
  end
```

```
  class Engine
    def cylinders
      6
    end
    def volume
      3
    end
    def power
      250
    end
  end
end
```

В методе `build` создается объект класса `Engine`, который помещается в инстанс-переменную `@engine`. Инстанс-переменные доступны для использования в других методах класса, поэтому в методе `engine` мы можем вернуть ее в качестве результата.

> ***Замечание***
>
> Инициализация инстанс-переменной `@engine` в методе `build` — не характерный прием разработки классов. Для инициализации используется метод `initialize` или метод-сеттер. Более детально эти методы рассмотрены в *главе 14*.

Для того чтобы воспользоваться классом, его можно включить в программу при помощи метода `require_relative` (листинг 13.15).

Листинг 13.15. Использование класса `Car`. Файл car_use.rb

```
require_relative 'car'

car = Car.new
car.build

puts car.title            # BMW X7
puts car.description      # Новый BMW X7 с окраской кузова...
puts car.engine.cylinders # 6
puts car.engine.volume    # 3
puts car.engine.power     # 250
```

Внутри класса `Car` можно обращаться к классу `Engine`, однако за пределами класса получить доступ к нему уже не получится — возникнет сообщение об ошибке:

```
 > require_relative 'car'
=> true
 > Car
=> Car
 > Engine
NameError: uninitialized constant Engine
```

Для получения доступа к вложенному классу `Engine` за пределами класса `Car` необходимо воспользоваться *оператором разрешения области видимости* `::`.

```
 > Car::Engine
=> Car::Engine
```

Сначала указывается класс в глобальной области видимости, затем оператор разрешения области видимости `::`, потом имя внутреннего класса. В листинге 13.16 приводится пример создания объекта двигателя в отрыве от автомобиля.

Листинг 13.16. Создание объекта двигателя в отрыве от автомобиля. Файл engine.rb

```
require_relative 'car'

engine = Car::Engine.new

puts engine.cylinders # 6
puts engine.volume    # 3
puts engine.power     # 250
```

В *разд. 13.5* было показано, что можно с легкостью открыть существующий класс, поправить его или даже сломать. Поэтому вложение классов имеет большое значение для организации *пространств имен*. Они используются для того, чтобы исключить конфликты одноименных классов из разных библиотек. Если потребуется ввести в программу корабли `Ship`, их двигатели `Ship::Engine` не будут конфликтовать с двигателями автомобилей `Car::Engine`.

Уровень вложения классов не ограничен. Наряду с классами в организации пространств имен участвуют модули (см. *главы 19–21*).

13.8. Константы

Названия классов `Car`, `Engine` или `Car::Engine` — это константы. Константа может ссылаться на любой объект, не только на класс. В листинге 13.17 в класс `Car::Engine` вводится три константы:

- `CYLINDERS` — количество цилиндров;
- `VOLUME` — объем двигателя;
- `POWER` — мощность двигателя.

Константы используются для замены цифр в одноименных методах.

Листинг 13.17. Использование констант в классе. Файл car_const.rb

```
class Car
  # ...
  def engine
    @engine
  end
```

```
  def build
    @engine = Engine.new
  end

  class Engine
    CYLINDERS = 6
    VOLUME = 3
    POWER = 250
    def cylinders
      CYLINDERS
    end
    def volume
      VOLUME
    end
    def power
      POWER
    end
  end
end
```

Получить значения констант можно как через методы объекта `engine`, так и напрямую — через оператор разрешения области видимости (листинг 13.18).

Листинг 13.18. Обращение к константам класса. Файл const.rb

```
require_relative 'car_const'

car = Car.new
car.build

puts car.engine.cylinders   # 6
puts car.engine.volume      # 3
puts car.engine.power       # 250

puts Car::Engine::CYLINDERS # 6
puts Car::Engine::VOLUME    # 3
puts Car::Engine::POWER     # 250
```

В отличие от инстанс-метода, для прямого обращения к константе, например `Car::Engine::CYLINDERS`, не требуется создание объекта класса.

13.9. Переменные класса

На уровне класса допускается использование *переменных класса*. Такая переменная начинается с двух символов `@@` и доступна во всех объектах класса. Кроме того, она доступна во всех методах класса (рис. 13.3).

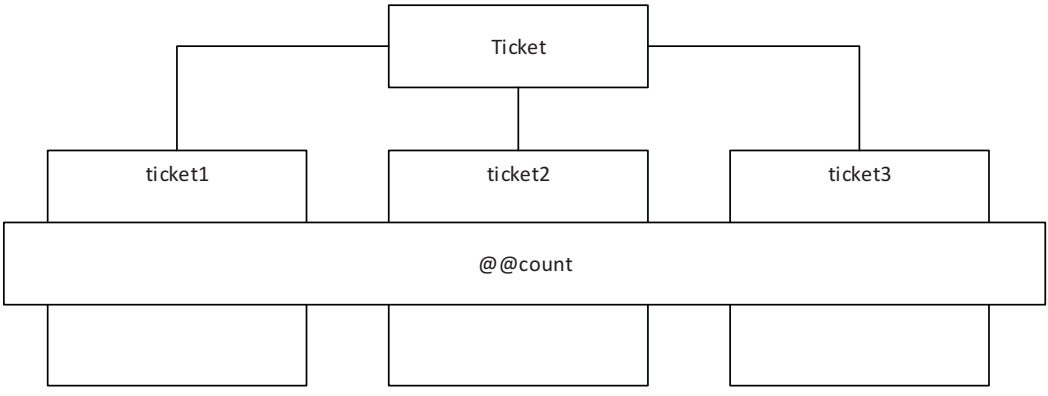


Рис. 13.3. Переменная класса `@@count`

Переменные класса удобно использовать для организации счетчиков и коллекций. В листинге 13.19 приводится пример класса завода `Factory`, в котором при помощи метода `Car` создается объект машины. Переменная класса `@@count` используется для подсчета количества созданных машин, а переменная `@@cars` содержит ссылку на все созданные в ходе работы программы объекты машин.

Листинг 13.19. Использование классовых переменных. Файл factory.rb

```
class Factory
  @@count = 0
  @@cars = []

  def build
    @@count += 1
    @@cars << Car.new
    @@cars.last
  end

  class Car
  end
end
```

Класс `Car` является вложенным классом для класса фабрики `Factory`. Для его создания используется метод `build`. При каждом обращении к методу значение переменной `@@count` увеличивается на единицу, а объект класса `Car` помещается в массив `@@cars`. Только что созданный объект машины извлекается из массива `@@cars` при помощи метода `last`. Эта последнее выражение в методе необходимо, чтобы метод `build` возвращал объект `Car.new`.

В листинге 13.20 приводится пример использования классов `Factory` и `Factory::Car`: создаются два завода (`Factory`), на которых создаются пять машин (`Car`).

Листинг 13.20. Файл factory_use.rb

```
require_relative 'factory'

puts 'Первый завод'
first = Factory.new
p first.build
p first.build

puts 'Второй завод'
second = Factory.new
p second.build
p second.build
p second.build

puts "Количество созданных машин #{first.count}"
puts 'Все созданные машины'
puts second.cars
```

В конце программы выводится отчет о количестве созданных машин и полный список объектов `Factory::Car`. Причем, для вызова методов `count` и `car` можно использовать как первую (`first`), так и вторую (`second`) фабрики. Каждый из объектов «знает» общее количество и список созданных машин за счет использования переменных класса. Результат работы программы может выглядеть следующим образом:

```
Первый завод
#<Factory::Car:0x007f7fc302b008>
#<Factory::Car:0x007f7fc302aea0>
Второй завод
#<Factory::Car:0x007f7fc302ac70>
#<Factory::Car:0x007f7fc302ab08>
#<Factory::Car:0x007f7fc302aa40>
Количество созданных машин 5
Все созданные машины
#<Factory::Car:0x007f7fc302b008>
#<Factory::Car:0x007f7fc302aea0>
#<Factory::Car:0x007f7fc302ac70>
#<Factory::Car:0x007f7fc302ab08>
#<Factory::Car:0x007f7fc302aa40>
```

Задания

1. Создайте класс здания `Building`. Какие методы в нем следует предусмотреть? Постройте объект этого класса и выведите его состояние.
2. Создайте класс шахмат, моделирующий шахматные фигуры. Какие методы в нем следует предусмотреть?

3. Создайте класс дней недели `Week`. Объекты класса должны отзываться на метод `each`, в блок которого передаются названия дней недели: понедельник, вторник, среда, четверг, пятница, суббота и воскресенье.
4. Пусть имеется фабрика, которая выпускает детские игрушки: плюшевый медвежонок (`TeddyBear`), мяч (`Ball`), кубики (`Cube`). Создайте класс `Factory`, который имеет метод `Factory.build`, возвращающий объект классов `TeddyBear`, `Ball` или `Cube`, соответствующий одной из игрушек. В качестве аргумента метод должен принимать один из символов: `:teddy_bear`, `:ball`, `:cube`, по которым будет приниматься решение о том, объект какого класса следует вернуть. В классе `Factory` следует предусмотреть метод `total`, который возвращает общее количество созданных игрушек. Кроме того, необходимо реализовать метод `offers`, возвращающий хэш, ключами которого выступают символы `:teddy_bear`, `:ball`, `:cube`, а значениями — количество созданных игрушек этого типа.
5. Добавьте в стандартный класс `Time` метод `hello`, который выводит приветствие, в зависимости от текущего времени суток. С 6:00 до 12:00 функция должна возвращать «Доброе утро», с 12:00 до 18:00 — «Добрый день», с 18:00 до 00:00 — «Добрый вечер», с 00:00 до 6:00 — «Доброй ночи».
6. Добавьте в стандартный класс `Integer` методы `minutes`, `hours`, `days`, которые возвращают количество секунд, соответствующих заданным значениям. Например, вызов `5.minutes` должен вернуть `300`, вызов `2.hours` — `7200`, а `1.days` — `86400`.
7. Создайте класс, объекты которого моделируют переход вещества в твердое (`solid`), жидкое (`liquid`) и газообразное состояние (`gaz`). Метод `status` должен возвращать состояние объекта. Кроме того, объекты класса должны поддерживать методы, которые переводят объект из одного состояния в другое:
   - `melt` — из твердого в жидкое;
   - `freeze` — из жидкого в твердое;
   - `boil` — из жидкого в газообразное;
   - `condense` — из газообразного в жидкое;
   - `sublime` — из твердого в газообразное;
   - `deposit` — из газообразного в твердое.
8. Решите предыдущую задачу при помощи гема `state_machine` (**https://github.com/pluginaweek/state_machine**).

ГЛАВА 14

Методы в классах

Файлы с исходными кодами этой главы находятся в каталоге *class_methods* сопровождающего книгу электронного архива.

Методы в классах имеют такие же свойства, как и глобальные методы. Они так же могут принимать параметры, уступать вычисления блокам, возвращать значения. Помимо приемов и свойств глобальных методов, с которыми мы познакомились в *главе 9*, в классах появляются новые, которые мы подробно рассмотрим в этой главе.

14.1. Сохранение состояния в объекте

Все объекты класса автоматически получают возможность использовать методы класса. В листинге 14.1 приводится пример класса `Ticket` с методом `price`, который возвращает цену билета.

Листинг 14.1. Инстанс-метод `price` в классе `Ticket`. Файл ticket_price.rb

```
class Ticket
  def price
    500
  end
end

first = Ticket.new
second = Ticket.new

puts first.price  # 500
puts second.price # 500
```

Каждый объект класса `Ticket` будет отзываться на метод `price`. Такие методы называются *инстанс-методами*, т. к. применять их можно только в отношении объектов класса.

Сейчас каждый билет имеет одну и ту же цену. Для назначения разным билетам разных цен необходимо, чтобы объекты могли сохранять состояние.

Чтобы сохранять в классе какое-либо состояние, необходимо воспользоваться инстанс-переменной. Такая переменная начинается с символа `@`. В листинге 14.2 в класс `Ticket` добавляются два инстанс-метода: `set_price` — для установки цены и `price` — для ее извлечения.

Листинг 14.2. Инстанс-переменная `@price` в классе `Ticket`. Файл ticket_instance.rb

```
class Ticket
  def set_price(price)
    @pirce = price
  end
  def price
    @pirce
  end
end

first = Ticket.new
second = Ticket.new

first.set_price(500)
second.set_price(600)

puts "Цена билета first: #{first.price}"
puts "Цена билета second: #{second.price}"
```

Метод `set_price` принимает единственный параметр `price`, который мы присваиваем инстанс-переменной `@price`. Начиная с этого момента, `@price` получает значение. Если обратиться к такой переменной до инициализации — переменная получит неопределенное значение `nil`:

```
 > @price
=> nil
```

Для получения доступа к переменной `@price` в класс `Ticket` вводится метод `price`, который просто возвращает `@price`. Вызов этого метода интерполируется в строку. В результате выполнения программы будут получены следующие строки:

```
Цена билета first: 500
Цена билета second: 600
```

Инстанс-переменные уникальны для каждого из объектов, т. е. в каждом объекте можно хранить свою цену билета.

14.2. Установка начального состояния объекта

Проблема программы из листинга 14.2 заключается в том, что между созданием объекта билета и установкой цены значение инстанс-переменной `@price` неопределено (листинг 14.3).

Листинг 14.3. Файл ticket_wrong.rb

```
...
ticket = Ticket.new
p ticket.price # nil

ticket.set_price(500)
p ticket.price # 500
```

14.2.1. Метод *initialize*

Для того чтобы задать объекту первоначальное непротиворечивое состояние, используется специальный метод `initialize`. Этот метод вызывается до всех остальных методов и позволяет инициализировать объект.

В листинге 14.4 создается объект `Ticket`, в котором цена билета по умолчанию составляет `500`.

Листинг 14.4. Установка цены по умолчанию. Файл ticket_initialize.rb

```
class Ticket
  def initialize
    puts 'Установка начального состояния объекта'
    @pirce = 500
  end

  def set_price(price)
    @pirce = price
  end

  def price
    @pirce
  end
end
```

В методе `initialize` цена билета устанавливается в размере 500 рублей. Кроме того, выводится фраза `'Установка начального состояния объекта'`, чтобы убедиться, что метод `initialize` будет вызван до всех остальных методов.

В листинге 14.5 приводится пример использования нового класса `Ticket`.

Листинг 14.5. Файл ticket_initialize_use.rb

```
require_relative 'ticket_initialize'

ticket = Ticket.new
puts "Цена билета: #{ticket.price}"
```

```
ticket.set_price(600)
puts "Цена билета: #{ticket.price}"
```

В результате выполнения программы выводятся следующие строки:

```
Установка начального состояния объекта
Цена билета: 500
Цена билета: 600
```

Таким образом, у нас не осталось участков в программе, где цена билета была бы неопределена. Объект билета сразу создается с ценой `500`, которую чуть позже можно изменить при помощи метода `set_price`.

Метод `initialize`, как и любой метод Ruby, может принимать параметры. Это означает, что цену билета можно назначать на этапе создания объекта. В листинге 14.6 приводится переработанный вариант класса `Ticket`, в котором цена билета устанавливается при создании объекта.

Листинг 14.6. Передача параметров методу `initialize`. Файл ticket_param.rb

```
class Ticket
  def initialize(price)
    @pirce = price
  end

  def price
    @pirce
  end
end
```

Теперь при создании объекта класса `Ticket` методу `new` необходимо передавать аргумент с ценой билета (листинг 14.7).

Листинг 14.7. Установка цены билета при создании объекта. Файл ticket_param_use.rb

```
require_relative 'ticket_param'

first = Ticket.new(500)
second = Ticket.new(600)

puts "Цена билета first: #{first.price}"
puts "Цена билета second: #{second.price}"
```

Если методу `initialize` задаются какие-либо параметры, методу `new` обязательно должно передаваться такое же количество аргументов. Если этого не сделать, создание объекта завершится ошибкой:

```
 > require_relative 'ticket_param'
=> true
```

```
 > ticket = Ticket.new
ArgumentError (wrong number of arguments (given 0, expected 1))
```

В тексте ошибки сообщается, что ожидается один аргумент, а передано — 0. Избежать этой ошибки можно, если параметру `price` в методе `initialize` задать значение по умолчанию:

```
def initialize(price = 500)
  @pirce = price
end
```

В таком случае можно вызывать метод `new` и с аргументом, и без него:

```
ticket = Ticket.new(500)
ticket = Ticket.new
```

Представим ситуацию, когда билету требуется больше атрибутов — например, необходимо задать дату и время сеанса. Для хранения даты потребуется еще одна инстанс-переменная `@date`. Кроме того, потребуются: метод `date` — для ее извлечения и расширение количества параметров, метод `initialize` — для корректной инициализации объекта (листинг 14.8).

Листинг 14.8. Добавляем дату и время проведения сеанса. Файл ticket_date.rb

```
class Ticket
  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end

  def price
    @pirce
  end

  def date
    @date
  end
end
```

В методе `initialize` используются позиционные параметры: `date` — для задания даты и времени и `price` — для цены. Параметр `price` имеет значение по умолчанию `500`, поэтому он размещен последним.

При создании объектов в методе `new` необходимо явно указывать имена позиционных параметров. В результате код получается читаемым и самодокументируемым (листинг 14.9).

Листинг 14.9. Файл ticket_date_use.rb

```
require_relative 'ticket_date'

first = Ticket.new(date: Time.mktime(2019, 5, 10, 10, 20))
```

```
puts "Дата билета first: #{first.date}"
puts "Цена билета first: #{first.price}"

second = Ticket.new(date: Time.mktime(2019, 5, 11, 10, 20), price: 600)

puts "Дата билета second: #{second.date}"
puts "Цена билета second: #{second.price}"
```

В приведенном примере создаются два билета: `first` и `second`. Дата и время проведения мероприятия задается при помощи метода `mktime` класса `Time`. В метод последовательно передаются год, месяц, день, час и минуты. В результате работы программы будут выведены следующие строки:

```
Дата билета first: 2019-05-10 10:20:00 +0300
Цена билета first: 500
Дата билета second: 2019-05-11 10:20:00 +0300
Цена билета second: 600
```

Количество инстанс-переменных в объекте не ограничено. Можно дальше продолжить совершенствование билета, добавляя название представления, ряд, место.

> ***Замечание***
>
> Ruby, помимо метода `initialize`, позволяет реализовать метод `initialize_copy`, который вызывается при клонировании объектов методами `dup` и `clone` (см. *разд. 5.3*).

14.2.2. Параметры метода *new*

В предыдущем разделе мы научились передавать параметры при создании объекта. Такой прием используется не только при создании своих собственных классов. Многие стандартные классы также позволяют задать начальное состояние своих объектов:

```
> 1..10
=> 1..10
> (1..10).class
=> Range
> Range.new 1, 10
=> 1..10
```

Здесь приводится пример создания диапазона (`Range`) от `1` до `10` при помощи метода `new`. Методу передаются аргументы: начало диапазона — `1` и его окончание — `10`.

В случае диапазона удобнее воспользоваться синтаксическим конструктором `1..10`. Однако не все объекты удобно создавать лишь синтаксическими конструкторами. Создать массив из десяти элементов проще при помощи явного вызова метода `new` класса `Array`. Для этого методу в качестве аргумента нужно передать число `10`:

```
 > Array.new 10
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]
```

В полученном массиве каждый элемент имеет неопределенное значение `nil`. Для инициализации массива каким-либо значением, например `0`, можно воспользоваться вторым необязательным параметром:

```
 > Array.new 10, 0
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

В качестве второго параметра можно передавать не только числа, но и строки:

```
 > Array.new 5, 'hello'
=> ["hello", "hello", "hello", "hello", "hello"]
```

В качестве инициализирующего значения может выступать любой объект:

```
 > Array.new 3, Object.new
=> [#<Object:0x00007fc574917c38>, #<Object:0x00007fc574917c38>,
                                  #<Object:0x00007fc574917c38>]
```

На рис. 14.1 показана ситуация, когда элементы полученного массива проинициализированы одним и тем же объектом.

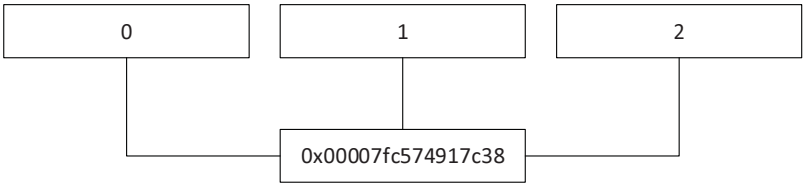


Рис. 14.1. Элементы массива ссылаются на один и тот же объект

Таким образом, при помощи `Object.new` был создан один объект, а все элементы массива проинициализированы ссылкой на этот единственный объект. Если будет изменен один объект массива, например первый, эти изменения получат все остальные элементы массива:

```
 > arr = Array.new 3, Object.new
=> [#<Object:0x00007fc5748c6ec8>, #<Object:0x00007fc5748c6ec8>,
#<Object:0x00007fc5748c6ec8>]
 > obj = arr.first
=> #<Object:0x00007fc5748c6ec8>
 > def obj.say; 'hello' end
=> :say
 > arr.last.say
=> "hello"
```

Здесь для первого элемента `arr.first` массива создается синглетон-метод `say`. Его можно вызывать и для последнего элемента `arr.last`.

Ситуация, когда все элементы массива ссылаются на один и тот же объект, не всегда подходит и может приводить к сложноулавливаемым ошибкам. Для инициализации массива разными объектами методу `new` массива можно передать блок. Блок будет вычисляться для каждого элемента индивидуально (рис. 14.2):

```
> arr = Array.new(3) { Object.new }
=> [#<Object:0x00007fc574818418>, #<Object:0x00007fc5748183c8>,
                                          #<Object:0x00007fc574818378>]
```

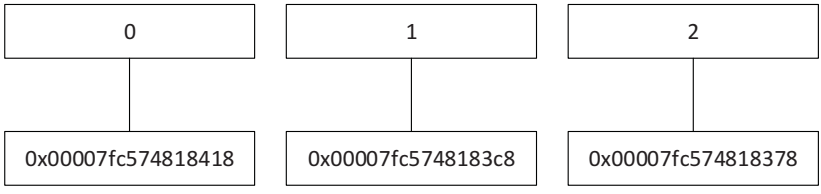


Рис. 14.2. Элементы массива ссылаются на разные объекты

Теперь элементы массива `arr` не зависят друг от друга:

```
 > obj = arr.first
=> #<Object:0x00007fc574818418>
 > def obj.say; 'hello' end
=> :say
 > obj.say
=> "hello"
 > arr.last.say
NoMethodError (undefined method `say' for #<Object:0x00007fc574818378>)
```

При помощи блоков можно инициализировать элементы массива разными значениями. В листинге 14.10 приводится пример программы, где элементы массива получают значения от `10` до `100` с шагом `10`.

Листинг 14.10. Инициализация элементов массива. Файл array_init.rb

```
n = 0
arr = Array.new(10) do
        n += 1
        n * 10
      end

p arr # [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Здесь локальная переменная `n` получает значение `0`, которое увеличивается в блоке на единицу. Блок — это псевдометод, который тоже возвращает значение. Последнее выражение блока становится возвращаемым результатом. Именно этот результат присваивается элементам массива.

Программу в листинге 14.10 можно попробовать усовершенствовать. Попробуем избавиться от инициализации `n = 0` и перенесем его внутрь блока:

```
 > Array.new(10) do
 >   n = 0
 >   n += 1
 >   n * 10
 > end
=> [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Каждый раз, когда вычисляется блок, переменная `n` получает нулевое значение, которое потом увеличивается на единицу и умножается на `10`. Как результат, все элементы массива получают одинаковые значения.

Переменную `n` нужно инициировать только первый раз, когда ее значение равно `nil`. Первое, что приходит в голову, — воспользоваться ключевым словом `unless`:

```
 > Array.new(10) do
 >   n = 0 unless n
 >   n += 1
 >   n * 10
 > end
=> [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Результат получился тем же самым, т. к. локальная переменная `n` не определена за пределами блока. Каждый раз, когда вычисляется содержимое блока, создается новая переменная `n` с неопределенным значением.

В глобальной области мы уже находимся внутри объекта. Следовательно, для сохранения значения от вызова к вызову мы можем воспользоваться инстанс-переменной `@n`:

```
 > Array.new(10) do
 >   @n = 0 unless @n
 >   @n += 1
 >   @n * 10
 > end
=> [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Теперь инициализация работает корректно. Если значение переменной `@n` будет равно `nil`, конструкция `unless` станет «рассматривать» его как `false`, в результате будет выполнено выражение `n = 0`.

Инициализацию переменной `@n` можно запрограммировать еще короче при помощи идиомы `||=`, а блок оформить в виде `proc` или `lambda` (листинг 14.11).

Листинг 14.11. Рефакторинг инициализации массива `arr`. Файл array_init_refactoring.rb

```
block = proc do
          @n ||= 0
          @n += 1
          @n * 10
        end
arr = Array.new(10, &block)

p arr # [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

14.2.3. Блоки в методе *new*

Блоки могут быть полезны для инициализации ваших собственных объектов. В листинге 14.12 приводится класс палитры цветов `Palette`.

Листинг 14.12. Класс палитры цветов `Palette`. Файл palette.rb

```
class Palette
  def initialize(colors = [])
    @colors = colors
  end

  def each
    @colors.each { |c| yield c }
  end
end

colors = %w[красный оранжевый желтый зеленый
            голубой синий фиолетовый]
pal = Palette.new(colors)
pal.each { |color| puts color }
```

Метод `initialize` принимает в качестве единственного необязательного параметра массив цветов. Этот массив помещается в инстанс-переменную `@colors`. В классе `Palette` реализуется метод `each`, позволяющий обойти цвета.

Модифицируем метод `initialize` таким образом, чтобы он мог принимать блок (листинг 14.13).

Листинг 14.13. Инициализация объекта блоком. Файл palette_block.rb

```
class Palette
  def initialize(colors = [])
    @colors = colors
    @colors = yield if block_given?
  end

  def each
    @colors.each { |c| yield c }
  end
end

pal = Palette.new do
  %w[красный оранжевый желтый зеленый
     голубой синий фиолетовый]
end
pal.each { |color| puts color }
```

В методе `initialize` при помощи метода `block_given?` проверяется, передан ли блок. Инстанс-переменная `@colors` инициализируется значением из параметра, если блок не задан. В случае использования блока результат его вычисления присваивается `@colors`.

14.2.4. Метод *initialize* и переменные класса

Метод `initialize` можно использовать не только для инициализации объекта. Этот метод идеально подходит для применения переменных класса. При помощи таких переменных можно собирать различную метаинформацию: коллекцию ссылок или количество созданных объектов (см. *разд. 13.8*).

Пусть имеется класс билета `Ticket`, и необходимо вести учет количества проданных билетов — например, чтобы не продать больше билетов, чем мест в зале (листинг 14.14).

Листинг 14.14. Использование переменной класса. Файл class_variable.rb

```
class Ticket
  @@counter = 0

  def initialize
    @@counter += 1
  end

  def counter
    @@counter
  end
end

first = Ticket.new
second = Ticket.new

puts first.counter # 2
```

Переменная класса `@@counter` получает начальное значение, равное `0`. Создание объекта сопровождается автоматическим вызовом метода `initialize`. В нем значение переменной `@@counter` увеличивается на единицу. Таким образом, переменная `@@counter` хранит количество созданных объектов `Ticket`.

В класс `Ticket` вводится еще один метод `counter`, чтобы добраться до значения переменной `@@counter` за пределами класса `Ticket`.

14.2.5. Как устроен метод *new*?

В других языках программирования метод, который инициализирует объект, называется *конструктором*. Имя конструктора часто совпадает либо с именем класса, либо с именем метода создания объекта (`new` — в случае Ruby).

Однако в Ruby метод инициализации называется `initialize`. Связано это с тем, что метод `new` вызывает два метода: `allocate` — для размещения объекта в памяти и метод `initialize`, который задает начальное состояние объекта:

```
def new
  allocate
  initialize
end
```

Метод `initialize` — это один из этапов, который выполняет метод `new`, прежде чем возвратить объект. Более того, если вместо метода `new` воспользоваться методом `allocate`, можно получить неинициализированный объект (листинг 14.15).

Листинг 14.15. Обход вызова `initialize`. Файл initialize.rb

```
class Ticket
  def initialize(price: 500)
    puts 'Установка начального состояния объекта'
    @pirce = price
  end

  def price
    @pirce
  end
end

ticket = Ticket.allocate
p ticket.price # nil
```

В приведенном примере метод `initialize` не вызывается.

14.3. Специальные методы присваивания

При работе с классами часто возникает ситуация, когда необходимо сохранить значение в инстанс-переменную, а затем его извлечь. Методы, устанавливающие значение, называются *сеттерами*, возвращающие — *геттерами*. В листинге 14.16 представлен класс билета, объект которого может сохранить дату представления `@date` и цену `@price`. Для обслуживания каждой из инстанс-переменных предусмотрены два метода. Метод с префиксом `set_` является сеттером. Метод, имя которого совпадает с именем инстанс-переменной, представляет собой геттер.

Листинг 14.16. Установка и извлечение цены и даты в билете. Файл ticket_set_get.rb

```
class Ticket
  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end
```

```
  def set_price(price)
    @pirce = price
  end
  def price
    @pirce
  end

  def set_date(date)
    @date = date
  end
  def date
    @date
  end
end
```

В объекте может быть множество инстанс-переменных. Каждая из них потребует и геттер, и сеттер. В результате в классе появляется множество однотипных методов. Язык программирования Ruby представляет разработчикам несколько механизмов для сокращения объема кода.

14.3.1. Методы со знаком равенства (=) в конце имени

Для сеттеров в Ruby предназначен специальный синтаксис — в конце имени метода допускается использование знака равенства. В листинге 14.17 названия методов `set_price` и `set_date` меняются на `price=` и `date=`.

Листинг 14.17. Использование сеттеров в классе Ticket. Файл ticket_setter.rb

```
class Ticket
  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end

  def price=(price)
    @pirce = price
  end
  def price
    @pirce
  end

  def date=(date)
    @date = date
  end
  def date
    @date
  end
end
```

Теперь инстанс-переменным `@price` и `@date` можно присваивать новые значения при помощи знака равенства (листинг 14.18).

Листинг 14.18. Файл ticket_setter_use.rb

```
require_relative 'ticket_setter'

ticket = Ticket.new(
           date: Time.new(2019, 5, 10, 10, 20),
           price: 500
         )

ticket.price = 600
ticket.date = Time.new(2019, 5, 11, 10, 20)

puts "Цена билета: #{ticket.price}"
puts "Билет на дату: #{ticket.date}"
```

В приведенном примере создается билет стоимостью `500` на 10 мая 2019 года. Затем цена билета увеличивается до `600`, а дата переносится на один день вперед. Вызов метода для установки новой цены должен выглядеть следующим образом:

```
ticket.price=(600)
```

Ruby позволяет не указывать круглые скобки вокруг аргументов:

```
ticket.price= 600
```

Кроме того, для сеттеров допускается размещать пробелы между именем и знаком равенства.

```
ticket.price = 600
```

Такое выражение выглядит более читаемым и привычным. Результатом выполнения программы будут следующие строки:

```
Цена билета: 600
Билет на дату: 2019-05-11 10:20:00 +0300
```

Сеттер не обязан присваивать значение инстанс-переменной. Его поведение может быть запрограммировано произвольно. В листинге 14.19 сеттер `price=` просто выводит значение параметра в стандартный поток вывода.

Листинг 14.19. Нетипичное использование сеттера. Файл setter.rb

```
class Setter
  def price=(price)
    puts price
  end
end
```

```
s = Setter.new
s.price = 500 # 500
```

Поведение метода `price=` в примере представляется запутывающим и нетипичным. В профессиональном коде сеттеры не используют для операций, не связанных с присваиванием. Это увеличивает время анализа кода и делает более дорогим его сопровождение.

14.3.2. Аксессоры

В листинге 14.17 класс `Ticket` имеет два геттера и два сеттера для обслуживания инстанс-переменных: цены `@price` и даты `@date`. Причем методы эти практически идентичны, изменяется только название методов и переменных.

Ruby предоставляет специальные методы класса, которые позволяют сократить объем кода. Речь идет об *аксессорах* — методах, которые позволяют создать другие методы.

Можно избавиться от методов `price` и `date`, если воспользоваться методом `attr_reader`. Метод принимает в качестве аргументов символы с названием инстанс-переменных (листинг 14.20).

Листинг 14.20. Использование метода `attr_reader`. Файл attr_reader.rb

```
class Ticket
  attr_reader :date, :price

  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end

  def price=(price)
    @pirce = price
  end

  def date=(date)
    @date = date
  end
end
```

Метод `attr_reader` создаст геттеры `price` и `date` и сколько угодно аналогичных дополнительных методов, если передать их названия в качестве аргументов.

При вызове метода `attr_reader` можно было бы использовать круглые скобки для того, чтобы подчеркнуть, что это метод:

```
attr_reader(:price, :date)
```

Однако в этом случае они намеренно опускаются, чтобы подчеркнуть декларативную природу метода. В этом случае `attr_reader` выглядит как объявление или клю-

чевое слово языка. Тем не менее, вы можете самостоятельно создавать и вызывать методы в теле класса (см. *разд. 14.5*).

Если воспользоваться методом `attr_writer`, можно избавиться от методов-сеттеров `price=` и `date=` (листинг 14.21).

Листинг 14.21. Использование метода `attr_writer`. Файл attr_writer.rb

```
class Ticket
  attr_reader :date, :price
  attr_writer :date, :price

  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end
end
```

В классе `Ticket` до сих пор имеются повторы: одинаковый набор аргументов для методов `attr_reader` и `attr_writer`. Их можно заменить одним методом `attr_accessor` (листинг 14.22).

Листинг 14.22. Использование метода `attr_accessor`. Файл attr_accessor.rb

```
class Ticket
  attr_accessor :date, :price

  def initialize(date:, price: 500)
    @pirce = price
    @date = date
  end
end
```

Метод `attr_accessor` создает геттер и сеттер для каждого переданного аргумента.

Язык программирования Ruby построен на соглашениях. Благодаря им можно разрабатывать компактные и элегантные программы. Небольшие команды программистов могут быстро и дешево создавать большие системы. Однако, чтобы это стало возможным, все соглашения Ruby должны постоянно присутствовать в головах у всех участников команды. Ни для кого из команды не должно быть какой-то непонятной магии в вызове методов `attr_accessor`. Все должны четко понимать, что в листинге 14.22 создаются четыре неявных метода, которыми можно воспользоваться. Следование соглашениям позволит создавать компактный и насыщенный код, легко умещающийся на одном экране монитора.

14.4. Синглетон-методы

Синглетон-метод — это метод, который определен непосредственно на объекте. Синглетон-методы существуют в единственном экземпляре и не могут быть вызваны без объекта, на котором они определены. При создании синглетон-метода перед именем метода указывается его получатель (рис. 14.3).

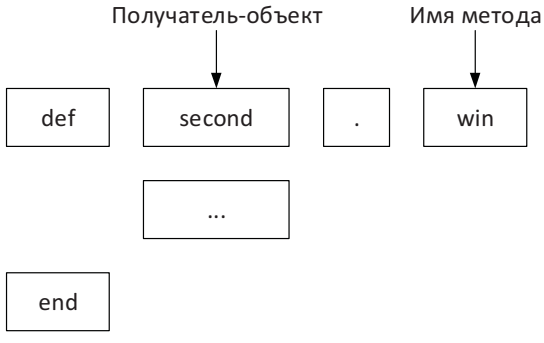


Рис. 14.3. Синглетон-метод

В листинге 14.23 приводится пример класса `Ticket`. Пусть среди зрителей, покупающих билеты, проводится лотерея. Второй билет `second` мы хотим назначить выигрышным. Для этого на объекте `second` создается синглетон-метод `win`, который будет признаком выигрышного билета.

Листинг 14.23. Создание синглетон-метода. Файл singleton_method.rb

```
class Ticket
  attr_accessor :price
  def initialize(price: 500)
    @price = price
  end
end

first = Ticket.new
second = Ticket.new(price: 600)

def second.win
  'Ваш билет выиграл'
end

puts "Цена билета first: #{first.price}"
puts first.win if first.respond_to? :win

puts "Цена билета second: #{second.price}"
puts second.win if second.respond_to? :win
```

Результатом работы программы будут следующие строки:

```
Цена билета first: 500
Цена билета second: 600
Ваш билет выиграл
```

За каждым объектом закреплено два класса: обычный и метакласс, который еще называют *engine-классом*. Обычный класс является общим для всех объектов этого класса, и в нем расположены инстанс-методы. Метакласс у каждого объекта свой собственный, и в нем хранятся синглетон-методы. Так как метакласс у каждого объекта свой, синглетон-методы не пересекаются, и каждый объект владеет своим собственным набором синглетон-методов (рис. 14.4).

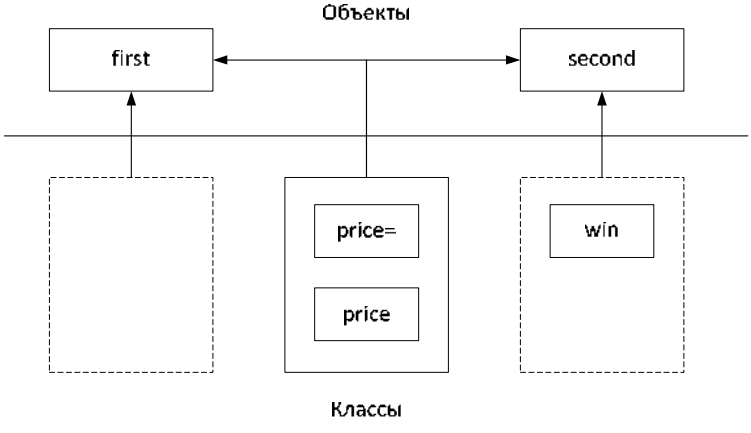


Рис. 14.4. Обычный класс и метаклассы объекта. Внизу прерывистой линией обозначены метаклассы объектов `first` и `second`. Сплошной линией обозначен класс `Ticket`

Метакласс, в отличие от обычного класса Ruby, не имеет константы-имени, тем не менее это тоже объект класса `Class`.

В *разд. 13.5* рассматривалось открытие класса. Каждый класс Ruby можно открыть и модифицировать. Не составляют исключение и метаклассы. Для их открытия предназначено ключевое слово `class`, после которого размещается оператор левого сдвига `<<`. В теле конструкции можно модифицировать метакласс объекта (рис. 14.5).

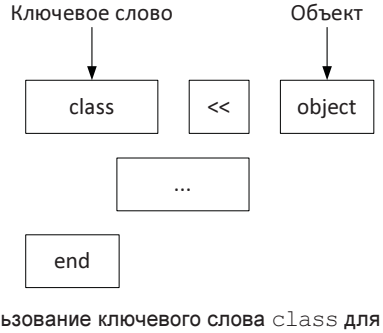


Рис. 14.5. Использование ключевого слова `class` для открытия класса

Рассмотрим пример с двумя билетами из листинга 14.23, в котором для создания метода `win` используется конструкция `class <<` (листинг 14.24).

Листинг 14.24. Использование конструкции `class <<`. Файл meta_class.rb

```
class Ticket
  attr_accessor :price
  def initialize(price: 500)
    @price = price
  end
end

first = Ticket.new
second = Ticket.new(price: 600)

class << second
  def win
    'Ваш билет выиграл'
  end
end

puts "Цена билета first: #{first.price}"
puts first.win if first.respond_to? :win

puts "Цена билета second: #{second.price}"
puts second.win if second.respond_to? :win
```

В метаклассе можно использовать аксессоры и любые другие методы (листинг 14.25).

Листинг 14.25. Файл meta_class_cost_and_attr.rb

```
ticket = Object.new

class << ticket
  attr_accessor :price, :date
end

ticket.price = 500
ticket.date = Time.mktime(2019, 5, 10, 10, 20)

puts "Дата билета: #{ticket.date}"
puts "Цена билета: #{ticket.price}"
```

Результатом выполнения программы будут строки:

```
Дата билета: 2019-05-10 10:20:00 +0300
Цена билета: 500
```

14.5. Методы класса

Классы тоже являются объектами. У них также есть общий класс `Class`, который содержит общие методы, — например, `new`. Кроме того, у каждого объекта класса есть метакласс, в котором могут располагаться синглетон-методы (рис. 14.6). В случае классов синглетон-методы называют *методами класса*.

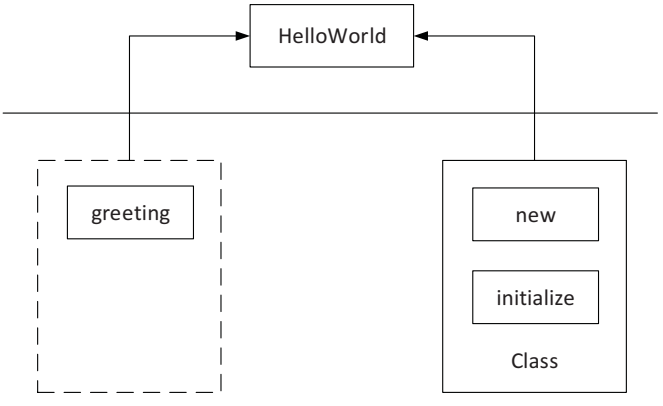


Рис. 14.6. Методы класса — это синглетон-методы объекта класса

В листинге 14.26 создается метод `greeting` класса `HelloWorld`. Для вызова такого метода не требуется создавать объект класса `HelloWorld`. У нас уже есть объект-получатель для вызова метода — `HelloWorld`.

Листинг 14.26. Создание метода класса. Файл class_method.rb

```
class HelloWorld
end

def HelloWorld.greeting
  'Hello, world!'
end

puts HelloWorld.greeting # Hello, world!
puts HelloWorld::greeting # Hello, world!
```

Для вызова метода можно использовать не только оператор «точка», но и оператор разрешения области видимости `::` . На практике оператор `::` применяется только для обращения к константам, а для методов всегда стараются использовать «точку».

Определение метода `greeting` можно поместить внутрь класса `HelloWorld` (листинг 14.27).

Листинг 14.27. Метод класса в теле класса. Файл class_method_inner.rb

```
class HelloWorld
  def HelloWorld.greeting
    'Hello, world!'
  end
end

puts HelloWorld.greeting # Hello, world!
```

Таким образом, в теле класса можно создавать два типа методов: класса и инстанс-методы (листинг 14.28).

Листинг 14.28. Метод класса и инстанс-методы. Файл method_instance_class.rb

```
class HelloWorld
  def HelloWorld.greeting
    'Метод класса'
  end

  def hello
    'Инстанс-метод, метод объекта'
  end
end

puts HelloWorld.greeting # Метод класса

say = HelloWorld.new
puts say.hello # Инстанс-метод, метод объекта
```

Метод `greeting` можно вызвать только в отношении класса `HelloWorld`, он отсутствует у объектов этого класса. Метод `hello` можно вызывать только у объектов класса. Попытка вызывать его у объекта `HelloWorld.hello` завершится ошибкой.

Имя класса `HelloWorld` внутри тела класса можно заменять ссылкой на текущий объект `self` (листинг 14.29).

Листинг 14.29. Использование ключевого слова `self`. Файл class_method_self.rb

```
class HelloWorld
  def self.greeting # HelloWorld.greeting
    'Hello, world!'
  end
end

puts HelloWorld.greeting # Hello, world!
```

Методы класса чаще всего встречаются именно в такой форме — с участием ключевого слова `self`.

Если название метода предваряется префиксом `self.` — это метод класса. Обычный метод означает, что перед нами синглетон-метод, который можно вызывать только в отношении объекта класса (рис. 14.7).

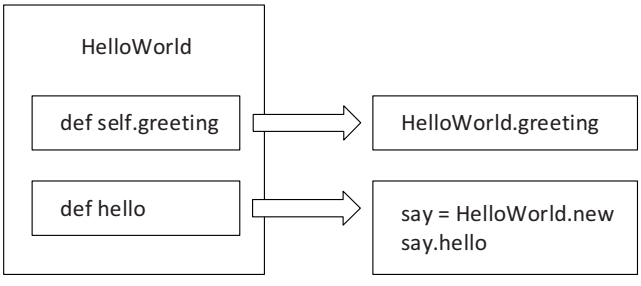


Рис. 14.7. Инстанс-методы и методы класса

В теле класса можно открыть метакласс и определить методы на его уровне. Это тоже будут методы класса, полностью аналогичные тем, которые получаются с использованием префикса `self.` (листинг 14.30).

Листинг 14.30. Использование конструкции `class <<`. Файл class_method_class.rb

```
class HelloWorld
  class << self
    def greeting
      'Hello, world!'
    end
  end
end

puts HelloWorld.greeting # Hello, world!
```

Как правило, префикс `self.` перед именем класса используется, когда необходимо добавить один метод класса. Конструкция `class <<` используется, когда методов класса много.

14.6. Обработка несуществующих методов

Ключевое слово `def` не единственный способ создать метод в объекте. Более того, мы уже убедились, что аксессоры могут создавать методы неявно. В этом разделе мы рассмотрим дополнительные способы создания методов, которые относятся к метапрограммированию. *Метапрограммирование* — это приемы, в которых программный код создает другой код.

14.6.1. Создание метода *define_method*

Создавать методы можно не только при помощи ключевого слова `def`, но и при помощи метода `define_method`, который тоже относится к приемам метапрограммирования (листинг 14.31).

Листинг 14.31. Создание метода `define_method`. Файл define_method.rb

```
class HelloWorld
  define_method :cube do |arg|
    arg ** 3
  end
end

hello = HelloWorld.new
puts hello.cube(2) # 8
```

В приведенном примере при помощи `define_method` создается метод `cube`. Имя будущего метода задается в виде символа `:cube`, который передается в качестве аргумента. В блоке, который принимает `define_method`, размещается реализация метода `cube`.

Методу `cube` передается аргумент `2`, подставляемый в параметр `arg`, который в теле метода возводится в куб.

Класс `HelloWorld`, представленный в листинге 14.31, аналогичен следующему классу:

```
class HelloWorld
  def cube(arg)
    arg ** 3
  end
end
```

Создание метода при помощи `define_method` выглядит запутаннее, им стараются не злоупотреблять, поскольку в большой программе становится сложным найти определение метода. К `define_method` прибегают тогда, когда необходимо сделать границу метода «прозрачной» для локальных переменных.

Кроме того, `define_method` полезен, когда необходимо создать сразу много однотипных методов, или названия методов заранее не известны. Например, при создании библиотеки взаимодействия с базой данных невозможно заранее предугадать названия таблиц и столбцов. В этом случае уместно использование приемов метапрограммирования — создания кода другим кодом.

Пусть имеется класс `Rainbow`, содержащий цвета радуги в хэше `COLORS`. В качестве ключа хэша выступают английские названия, в качестве значений — русские. При помощи `define_method` можно в цикле определить метод для каждого цвета. Английские названия цветов будут выступать названиями методов, обращение к которым станет возвращать русское название цвета (листинг 14.32).

Листинг 14.32. Массовое создание методов. Файл rainbow.rb

```
class Rainbow
  COLORS = {
    red: 'красный',
    orange: 'оранжевый',
    yellow: 'желтый',
    green: 'зеленый',
    blue: 'голубой',
    indigo: 'синий',
    violet: 'фиолетовый'
  }

  COLORS.each do |method, name|
    define_method method do
      name
    end
  end
end

r = Rainbow.new
puts r.yellow # желтый
puts r.red    # красный
```

Здесь элементы хэша `COLORS` перебираются итератором `each`. Блок итератора `each` принимает два параметра: ключ хэша `method` — с названием цвета на английском языке и его значение `name` — с названием метода на русском. Эти параметры используются в вызове `define_method`, чтобы сформировать семь методов — по количеству цветов радуги.

При использовании ключевого слова `def` методу невозможно было бы задать динамическое имя, кроме того, локальную переменную `name` нельзя было бы передать внутрь метода.

14.6.2. Перехват вызовов несуществующих методов

Ruby позволяет обрабатывать несуществующие методы. Для этого предназначен специальный перехватчик `method_missing`. Достаточно реализовать его в классе, чтобы все обращения к несуществующим методам для объектов этого класса перехватывались `method_missing` (листинг 14.33).

Листинг 14.33. Перехват вызовов несуществующих методов. Файл method_missing.rb

```
class HelloWorld
  def method_missing(m)
    puts m
```

```
    puts 'Такого метода нет'
  end
end

hello = HelloWorld.new
hello.world
```

В программе создается объект класса `HelloWorld`, у которого вызывается несуществующий метод `world`. Обращение к этому методу приводит к срабатыванию `method_missing`. Метод принимает один обязательный аргумент — название метода. Результатом работы программы будут следующие строки:

```
world
Такого метода нет
```

Если важно получить аргументы несуществующего метода, при создании `method_missing` можно передать необязательный параметр под аргументы. Лучше предварить этот параметр оператором `*`, чтобы иметь возможность принимать неограниченное количество параметров (листинг 14.34).

Листинг 14.34. Файл method_missing_args.rb

```
class HelloWorld
  def method_missing(m, *args)
    print 'Название метода: '
    puts m
    print 'Аргументы: '
    p args
  end
end

hello = HelloWorld.new
hello.world(42, 'set', 'get')
```

Результатом работы программы будут следующие строки:

```
Название метода: world
Аргументы: [42, "set", "get"]
```

Можно переработать класс `Rainbow` (листинг 14.32), используя `method_missing` (листинг 14.35).

Листинг 14.35. Файл rainbow_method_missing.rb

```
class Rainbow
  COLORS = {
    red: 'красный',
    orange: 'оранжевый',
    yellow: 'желтый',
```

```
    green: 'зеленый',
    blue: 'голубой',
    indigo: 'синий',
    violet: 'фиолетовый'
  }

  def method_missing(name)
    COLORS[name]
  end
end

r = Rainbow.new
puts r.yellow # желтый
puts r.red    # красный
```

В приведенном примере обращение к несуществующему методу приводит к вызову `method_missing`. В параметр `name` подставляется название несуществующего метода. После чего производится попытка извлечения значения из хэша `COLORS` значения с таким ключом. Если значение находится, возвращается русское название цвета, иначе возвращается неопределенное значение `nil`.

Метод `method_missing` тоже относится к приемам метапрограммирования. Организация методов при помощи `define_method` зачастую более эффективна, по сравнению с `method_missing`. Это связано с особенностями поиска метода (см. *главу 20*).

14.7. Метод *send*

В предыдущем разделе было показано, что имена методов часто могут быть динамическими и неизвестны разработчику заранее. Иногда мы не можем предугадать имя метода, а следовательно, не можем запрограммировать его вызов традиционным способом через оператор «точка».

В этом случае на помощь приходит метод `send`, который принимает в качестве параметра название метода и осуществляет его вызов (листинг 14.36).

> ***Замечание***
>
> Слово `send` весьма часто используется для пользовательских методов. В этом случае доступ к методу `send` может быть перекрыт, т. к. будет вызываться переопределенный метод `send`. На этот случай в Ruby предусмотрен псевдоним метода с двумя начальными и завершающими символами подчеркивания: `__send__`.

Листинг 14.36. Использование метода `send`. Файл send.rb

```
class HelloWorld
  def greeting
    'Hello, world!'
  end
end
```

```
h = HelloWorld.new
puts h.send(:greeting) # Hello, world!
```

Здесь можно было бы обойтись прямым вызовом метода `greeting`:

```
puts h.greeting
```

Более того, в этом случае код получается проще и компактнее. Однако бывают ситуации, когда без метода `send` обойтись трудно.

Давайте разработаем класс `Settings`, при создании объекта которого будет ожидаться блок. В блоке будут приниматься произвольные параметры. Названия параметров выступают в качестве методов объекта (листинг 14.37).

Листинг 14.37. Использование метода `send`. Файл settings_use.rb

```
require_relative 'settings'

settings = Settings.new do |s|
  s.hello = 'world'
  s.page = 1
  s.number = 30
end

puts settings.hello  # world
puts settings.page   # 1
puts settings.number # 30
```

Так как конструктор класса `Settings` принимает блок, необходимо вызвать в методе `initialize` ключевое слово `yield`:

```
class Settings
  def initialize
    yield @obj
  end
  def method_missing(name)
    @obj.send(name) if @obj.respond_to? name
  end
end
```

Здесь блок принимает в качестве параметра объект. Пока мы не знаем, что это за объект, нам предстоит его разработать. Так как объект `Settings` должен отзываться на имена методов, переданных в параметре, необходимо реализовать метод `method_missing`, который перехватит такие обращения. Переадресовать обращения мы можем к методу `@obj`, т. к. только он «знает», какие параметры были введены в блоке.

Для вызова метода используется метод `send`, т. к. название метода поступает нам в виде параметра `name`. Использовать для вызова оператор «точка» не получится.

Если использовать класс `Settings` в таком виде, программа завершится ошибкой, поскольку объект `@obj` не инициализирован и принимает значение `nil`. Очевидно, необходимо что-то сделать с объектом `@obj`, который должен быть гораздо «умнее», чем обычный объект Ruby.

Давайте разработаем для него класс `Storage`. В листинге 14.38 приводится заготовка класса.

Листинг 14.38. Заготовка класса `Storage`. Файл storage_init.rb

```
class Storage
  attr_accessor :params

  def initialize
    @params = {}
  end
end
```

В классе `Storage` заводим инстанс-переменную `@params`, в которой сохраняем переданные пользователем параметры. Сразу после создания эта переменная инициализируется пустым хэшем.

Создадим объект этого класса `s`, причем нам необходимо добиться от него возможности добавления произвольного параметра `s.hello = 'world'`. При этом в инстанс-переменной `@params` должна появляться новая пара «ключ-значение» (листинг 14.39).

Листинг 14.39. Использование класса `Storage`. Файл storage_init_use.rb

```
require_relative 'storage_init'

s = Storage.new
s.hello = 'world'
p s.params
```

Сейчас нужного поведения у класса `Storage` нет. Если запустить программу на выполнение, будет получено сообщение об ошибке:

```
undefined method `hello=' (NoMethodError)
```

Невозможно добавить для каждого произвольного параметра свой собственный сеттер. Мы просто не сможем предугадать, какие сеттеры могут потребоваться пользователю. Поэтому следует перехватить такие обращения при помощи `method_missing`:

```
class Storage
  attr_accessor :params

  def initialize
    @params = {}
  end
```

```
  def method_missing(name, *args)
    p name
  end
end
```

В method_missing перехватываются не только названия метода name, но и его параметры args. Это необходимо, т. к. у сеттера всегда имеется параметр — присваиваемое значение.

Имя name для сеттеров представляет собой символы вида:

```
:name=
:page=
:number=
```

Для того чтобы сформировать ключ для параметра, символ «равно» в конце названия необходимо отрезать. Однако символы — неизменяемые объекты, их нельзя преобразовывать. Поэтому символы при помощи метода to_s необходимо перевести в строку, которую уже можно изменять (листинг 14.40).

Листинг 14.40. Класс Storage. Файл storage.rb

```
class Storage
  attr_accessor :params

  def initialize
    @params = {}
  end

  def method_missing(name, *args)
    method = name.to_s
    return unless method.end_with? '='
    @params[method.chomp('=').to_sym] = args.first
  end
end
```

Объект класса Storage реагирует только на сеттеры. Поэтому, если в конце метода нет символа «равно», method_missing ничего не предпринимает и завершает работу. Проверяем наличие «равно» в конце имени метода при помощи специального метода end_with?. Если проверка завершается неудачно, покидаем метод при помощи ключевого слова return.

В качестве ключа хэша @params используется название метода без завершающего знака «равно». Из параметров извлекаем первый при помощи метода first. Класс Storage готов к использованию (листинг 14.41).

Листинг 14.41. Использование класса `Storage`. Файл storage_use.rb

```
require_relative 'storage_init'

s = Storage.new
s.hello = 'world'
p s.params
```

Результатом выполнения программы будут следующие строки:

```
{"hello"=>"world", "page"=>1, "number"=>30}
```

Теперь пришло время вернуться к классу `Settings`. Объект `@obj` можно инициализировать новым объектом класса `Storage` (листинг 14.42).

Листинг 14.42. Класс `Settings`. Файл settings.rb

```
require_relative 'storage'

class Settings
  def initialize
    @obj = Storage.new
    yield @obj
  end
  def method_missing(name)
    @obj.params[name]
  end
end
```

Вызов `send` можно заменить на прямое обращение к хэшу `params`. Теперь запуск исходной программы будет приводить к ожидаемым результатам — объект `settings` станет реагировать на методы, заданные в его блоке при инициализации:

```
require_relative 'settings'

settings = Settings.new do |s|
  s.hello = 'world'
  s.page = 1
  s.number = 30
end

puts settings.hello  # world
puts settings.page   # 1
puts settings.number # 30
```

Задания

1. Создайте класс `Hello`, при создании объекта которого конструктору можно было бы передавать параметр `Hello.new('world')`. При вызове метода `say` этого объекта `hello.say` или при выводе объекта методом `puts` выводилась бы фраза `'Hello, world!'`.
2. Создайте класс пользователя `User`, который бы при создании объекта позволял задавать фамилию, имя и отчество. Метод должен реализовывать методы чтения и установки фамилии, имени и отчества.
3. Создайте класс `Group`, конструктор которого может принимать произвольное количество объектов класса `User` (из предыдущего задания). Реализуйте в классе `Group` метод `each`, при помощи которого можно обойти список пользователей, — например, чтобы вывести их в стандартный поток.
4. Создайте класс `Foo`, метод `new` которого может принимать хэш. Объекты класса `Foo` должны отзываться на методы, названия которых совпадают с ключами хэша. В качестве результата методы должны возвращать значения, соответствующие ключу в хэше.
5. Создайте класс `List`, метод `new` которого может принимать произвольное количество параметров. Объекты класса должны поддерживать метод `each`, в блок которого последовательно передаются параметры, заданные при создании объекта.

ГЛАВА 15

Преобразование объектов

Файлы с исходными кодами этой главы находятся в каталоге *overload* сопровождающего книгу электронного архива.

В Ruby приходится постоянно иметь дело с объектами и программировать взаимодействие объектов. Объекты можно складывать, интерполировать в строку, обращаться при помощи квадратных скобок к их составным частям. Все эти действия совершаются при помощи методов и операторов (см. *главу 7*). Иногда методы объекта вызываются неявно — например, как это происходит в случае интерполяции объекта в строку.

Операторы являются методами, которые можно заменить своими собственными методами. Можно повторно открыть класс или модифицировать объект, назначить операторам свое собственное поведение. Наличие того или иного метода в классе или объекте может превратить его в строку или в массив. В этой главе мы подробнее остановимся на перегрузке операторов и преобразовании объектов друг в друга.

15.1. Сложение строк и чисел

Оператор сложения интуитивно понятен: из двух объектов мы получаем один объект побольше. При сложении двух чисел получается их сумма, при сложении строк — объединенная строка:

```
 > 7 + 5
=> 12
 > 'hello' + ' ' + 'ruby'
=> "hello ruby"
```

Чем сложнее операция, тем больше альтернативных путей. Например, объединить три последние строки можно, объединив их в массив и воспользовавшись методом `join`:

```
 > ['hello', ' ', 'ruby'].join
=> "hello ruby"
 > ['hello', 'ruby'].join(' ')
=> "hello ruby"
```

Более того, методу join можно передать в качестве аргумента строку-разделитель. В приведенном примере — это пробел. Методы предоставляют гораздо более гибкие и широкие возможности. Однако они менее наглядные. По наличию оператора + можно догадаться, что он делает без документации и экспериментов в консоли. После нескольких лет школьного курса трудно забыть, как работает оператор «плюс».

Догадаться, что делает метод join уже труднее, даже если вы хорошо знаете английский язык. Особенно, если метод принимает параметры или блоки.

Поэтому в объектно-ориентированных языках операторам уделяется особое внимание. Однако они часто подводят, если слева и справа от оператора располагаются объекты разной природы. Например, нельзя складывать числа и строки:

```
> 3 + '2'
TypeError (String can't be coerced into Integer)
> 'tv' + 360
TypeError (no implicit conversion of Integer into String)
```

Ruby-интерпретатор не может здесь догадаться, какой результат мы хотим получить: привести строку '2' к числу и получить значение 5 или привести число 360 к строке и получить 'tv360'.

15.2. Методы преобразования объектов

В рассмотренном только что случае потребуется явное приведение объектов к числовому или строковому значениям:

```
 > 3 + '2'.to_i
=> 5
 > 'tv' + 360.to_s
=> "tv360"
```

Методы to_i и to_s нам уже встречались в предыдущих главах (см. *разд. 4.5* и *разд. 7.4.2*). Кроме того, в *разд. 4.3* упоминался метод to_sym для преобразования объектов в символ.

Методы преобразования, как правило, уже реализованы в объектах. Наиболее распространенные приводятся в табл. 15.1.

Таблица 15.1. *Методы преобразования объектов*

Метод	Описание
to_s	Преобразование к строке (String)
to_i	Преобразование к целому числу (Integer)
to_f	Преобразование к числу с плавающей точкой (Float)
to_sym	Преобразование к символу (Symbol)
to_a	Преобразование к массиву (Array)
to_h	Пребразование к хэшу (Hash)

Метод `to_f` позволяет преобразовать строку или целое число к числу с плавающей точкой:

```
 > '36.6'.to_f
=> 36.6
 > 2.to_f
=> 2.0
```

Метод `to_a` позволяет преобразовать объект в массив. Не каждый объект может быть преобразован в массив. Для этого объект должен быть достаточно сложным. Например, из хэша получается массив массивов:

```
 > { fst: :hello, snd: :world }.to_a
=> [[:fst, :hello], [:snd, :world]]
```

Каждый элемент такого массива представляет собой массив из двух элементов: ключа и значения. Разложить в массив при помощи метода `to_a` можно объект класса `Time`:

```
 > t = Time.mktime(2019, 5, 11, 10, 20)
=> 2019-05-11 10:20:00 +0300
 > t.to_a
=> [0, 20, 10, 11, 5, 2019, 6, 131, false, "MSK"]
```

В результирующем массиве объект класса времени `Time` раскладывается на отдельные составляющие: секунды, минуты, часы, дни, месяц, год, день недели, день года, зимнее время и часовой пояс.

Метод `to_h` может быть применен только к массивам специального типа. Каждый элемент в нем должен быть массивом из двух элементов: ключа и значения:

```
 > [[:fst, :hello], [:snd, :world]].to_h
=> {:fst=>:hello, :snd=>:world}
```

Метод `to_h` выступает обратным преобразованием для метода `Hash#to_a`.

Многие классы, помимо методов с префиксом `to_`, реализуют дополнительные методы преобразования. Частыми операциями являются преобразование коллекции в строку и, наоборот, преобразование строки в коллекцию.

Метод `join` преобразует массив в строку. В качестве необязательного параметра метод может принимать строку-разделитель, которая будет вставлена между элементами массива. Если аргумент не указывается, разделителем выступает пустая строка, и элементы массива просто следуют друг за другом (листинг 15.1).

Листинг 15.1. Использование метода `join`. Файл join.rb

```
p [1, 2, 3, 4, 5].join                        # "12345"
p [1, 2, 3, 4, 5].join('-')                   # "1-2-3-4-5"
p ['Сергей', 'Петрович', 'Иванов'].join(' ') # "Сергей Петрович Иванов"
```

Метод `join` справляется не только с линейными, но и с вложенными массивами (листинг 15.2).

Листинг 15.2. Объединение элементов вложенного массива. Файл join_nested.rb

```
p [1, 2, [3, 4, 5], ['a', 'b']].join('-') # "1-2-3-4-5-a-b"
```

Метод `split` является обратным методу `join`. Его используют для преобразования строки в массив (листинг 15.3).

Листинг 15.3. Преобразование строки в массив. Файл split.rb

```
p 'Сергей Петрович Иванов'.split # ["Сергей", "Петрович", "Иванов"]
```

По умолчанию в качестве разделителя выступает пустая строка, однако разделитель можно задать при помощи необязательного аргумента (листинг 15.4).

Листинг 15.4. Явное задание разделителя. Файл split_separator.rb

```
p '1-2-3-4-5'.split('-') # ["1", "2", "3", "4", "5"]
```

В результирующем массиве элементы являются строками, и чтобы получить числа, необходимо преобразовать элементы при помощи метода `to_i`. Для этого можно воспользоваться итератором `map` (листинг 15.5).

Листинг 15.5. Преобразование элементов к целому числу. Файл split_map.rb

```
p '1-2-3-4-5'.split('-').map(&:to_i) # [1, 2, 3, 4, 5]
```

Метод `split` может принимать блок, в который последовательно передает извлеченные из строки элементы (листинг 15.6).

Листинг 15.6. Использование блока совместно с методом `split`. Файл split_block.rb

```
result = '1-2-3-4-5'.split('-') { |x| p x.to_i }
p result # "1-2-3-4-5"
```

Результатом выполнения программы будут следующие строки:

```
1
2
3
4
5
"1-2-3-4-5"
```

При использовании блока метод возвращает исходную строку, а не массив.

Метод `split` может принимать второй необязательный параметр, который ограничивает количество элементов в результирующем массиве (листинг 15.7).

Листинг 15.6. Ограничение размера результирующего массива. Файл split_limit.rb

```
p '1-2-3-4-5'.split('-', 1) # ["1-2-3-4-5"]
p '1-2-3-4-5'.split('-', 3) # ["1", "2", "3-4-5"]
```

В строке могут встречаться участки, где разделители следуют один за другим. Если такие последовательности встречаются в середине строки, создается элемент с пустой строкой. В конце строки последовательность из нескольких разделителей игнорируется (листинг 15.7).

Листинг 15.7. Несколько подряд идущих разделителей. Файл split_negative.rb

```
p '1--2--3--4-5---'.split('-')  # ["1", "", "2", "", "3", "", "4", "5"]
p '1-2-3-4-5---'.split('-')     # ["1", "2", "3", "4", "5"]
p '1-2-3-4-5---'.split('-', -1) # ["1", "2", "3", "4", "5", "", "", ""]
```

Если в качестве второго параметра передается отрицательное значение, параметр перестает регулировать размер результирующего массива. Вместо этого меняется поведение метода `split` в отношении завершающих разделителей: они учитываются при формировании результата. Числовое значение аргумента при этом не имеет значения — с одинаковым успехом можно использовать как `-1`, так и `-100`.

Для более сложных преобразований можно реализовывать свои собственные методы-конвертеры. В листинге 15.8 приводится пример разбора строки:

```
'2019-10-23 15:20:06'
```

из которой извлекаются год, месяц, день, час, минуты и секунды. В качестве результата преобразования получается хэш, в котором в качестве ключей выступают символы `:year`, `:month`, `:day`, `:hours`, `:minutes` и `:seconds`.

Листинг 15.8. Преобразование строки-даты в хэш. Файл split_date.rb

```
def split_date(str)
  date, time = str.split
  year, month, day = date.split('-', 3).map(&:to_i)
  hours, minutes, seconds = time.split(':', 3).map(&:to_i)

  {
    year: year,
    month: month,
    day: day,
    hours: hours,
    minutes: minutes,
    seconds: seconds
  }
end

p split_date('2019-10-23 15:20:06')
```

В качестве результата выполнения программы будет возвращен следующий хэш:

```
{:year=>2019, :month=>10, :day=>23, :hours=>15, :minutes=>20, :seconds=>6}
```

Для удобства повторного использования кода преобразования строки в хэш реализован метод `split_date`. В нем строка сначала разбивается по пробелу, в результате чего получаются две переменные: `date` и `time`, которые в свою очередь также разбиваются методом `split` на отдельные компоненты. Причем каждый элемент дополнительно преобразуется к целому числу при помощи метода `to_i`.

В конце метода при помощи фигурных скобок формируется объект класса `Hash`. Так как это последнее выражение метода, полученный хэш будет возвращаться в качестве результата метода.

В рассматриваемом случае это именно хэш, а не блок метода. Чтобы не путаться, между открывающей фигурной скобкой и предыдущим выражением добавляется пустая строка.

15.3. Сложение объектов

Пусть имеется класс билета `Ticket`, в котором при помощи `attr_accessor` определены геттер и сеттер для инстанс-переменной `@price`. Кроме того, задан метод `initialize`, позволяющий установить значение этой переменной при создании объекта (листинг 15.9).

Листинг 15.9. Заготовка для класса билета `Ticket`. Файл ticket_stub.rb

```
class Ticket
  attr_accessor :price

  def initialize(price)
    @price = price
  end
end
```

Попытка сложения двух билетов проваливается — объекты класса `Ticket` не поддерживают оператор `+`:

```
 > require_relative 'ticket_stub'
=> true
 > Ticket.new(500) + Ticket.new(600)
NoMethodError (undefined method `+' for #<Ticket:0x00007fc461162830
                                                        @price=500>)
```

Интерпретатор Ruby не «знает», по какой логике должны складываться объекты пользователей и какой результат должен получиться в конце. Будет это массив объектов, строка или сумма цен? Решить может только разработчик, только он знает назначение класса `Ticket`, и как его объекты должны себя вести. Модифицируем класс, чтобы билеты поддерживали сложение. Пусть результатом будет сумма цен билетов. Для этого в классе `Ticket` определим метод с именем `+` (листинг 15.10).

Листинг 15.10. Перегрузка оператора +. Файл ticket.rb

```
class Ticket
  attr_accessor :price

  def initialize(price)
    @price = price
  end

  def +(ticket)
    price + ticket.price
  end
end
```

Теперь сложение объектов класса `Ticket` происходит так, как мы задумали:

```
 > require_relative 'ticket'
=> true
 > Ticket.new(500) + Ticket.new(600)
=> 1100
```

15.4. Сложение объекта и числа

Расширим пример из предыдущего раздела таким образом, чтобы сложение двух объектов возвращало сумму цен — точно так же, как в листинге 15.10. Сложение объекта с числом будет приводить к созданию нового объекта с увеличенной ценой. Для этого переопределим метод `+` (листинг 15.11).

Листинг 15.11. Сложение объекта билета с числом. Файл ticket_plus_number.rb

```
class Ticket
  attr_accessor :price

  def initialize(price)
    @price = price
  end

  def +(value)
    case value
    when Ticket
      price + value.price
    when Numeric
      ticket = self.dup
      ticket.price += value
      ticket
    end
  end
end
```

Для того чтобы определить в методе +, какой аргумент был передан, используется конструкция case. В when-условие передается класс, с которым сравнивается параметр value. Сравнения в примере аналогичны использованию оператора ===:

```
 > Ticket === 200
=> false
 > Numeric === 200
=> true
```

Вместо классов Integer или Float используется класс Numeric. Это позволяет охватить оба случая: и целые, и вещественные числа. Такой прием срабатывает в силу того, что классы Integer и Float являются производными классами Numeric (рис. 15.1). Более подробно наследование рассматривается в *главе 17*.

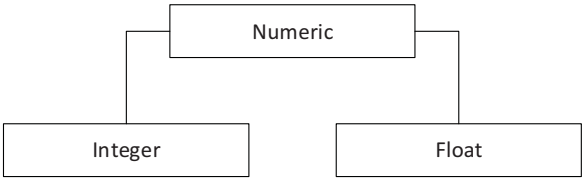


Рис. 15.1. Классы Integer и Float являются наследниками общего класса Numeric

В случае сложения билета с объектом класса Ticket будут сложены их цены: price + value.price.

Если в качестве правого аргумента взять число, то будет сформирован новый объект билета. Чтобы не изменять текущий объект, при помощи метода dup создается копия объекта self (ключевое слово self является ссылкой на текущий объект и детально рассматривается в *главе 16*).

При сложении мы не привыкли, чтобы слагаемые изменялись, поэтому не будем нарушать традицию. Новый объект помещаем в локальную переменную ticket, увеличиваем его цену на число value и возвращаем новый объект в качестве результата:

```
 > require_relative 'ticket_plus_number'
=> true
 > Ticket.new(500) + Ticket.new(600)
=> 1100
 > t = Ticket.new(500)
=> #<Ticket:0x00007f9e1b9a86f0 @price=500>
 > t + 10.0
=> #<Ticket:0x00007f9e1b9b1458 @price=510.0>
 > t
=> #<Ticket:0x00007f9e1b9a86f0 @price=500>
 > t + 200
=> #<Ticket:0x00007f9e1b970c78 @price=700>
 > t
=> #<Ticket:0x00007f9e1b9a86f0 @price=500>
```

Как видно из результатов, сложение не приводит к изменению исходного объекта. Чтобы мы ни делали, его цена остается равной `500`. В ходе сложения получается новый билет с новой ценой.

В случае, если нам потребуется метод сложения, в ходе которого исходный объект изменяется, лучше реализовать это bang-методом `add!` (листинг 15.12). Восклицательный знак будет сигнализировать о том, что в ходе выполнения метода состояние объекта изменится.

Листинг 15.12. Реализация bang-метода `add!`. Файл ticket_add_bang.rb

```
class Ticket
  attr_accessor :price

  def initialize(price)
    @price = price
  end

  def +(value)
    case value
    when Ticket
      price + value.price
    when Numeric
      add(value)
    end
  end

  def add(value)
    ticket = self.dup
    ticket.price += value
    ticket
  end

  def add!(value)
    @price += value
    self
  end
end
```

В новом классе `Ticket` создаются два метода: безопасный `add`, который не изменяет состояние объекта, и `add!`, приводящий к изменению цены текущего билета. Для того чтобы не дублировать код, выражения в `when`-условии заменяются на вызов метода `add`:

```
 > require_relative 'ticket_add_bang'
=> true
 > t = Ticket.new(500)
=> #<Ticket:0x00007fb704066568 @price=500>
```

```
 > t.add!(10)
=> #<Ticket:0x00007fb704066568 @price=510>
 > t + 200
=> #<Ticket:0x00007fb704077cc8 @price=710>
```

15.5. Сложение объекта и строки

Помимо сложения, объекты поддерживают интерполяцию в строку. Для этого объект помещают в последовательность `#{}`. Интерполяция неявно вызывает метод `to_s`. Выражения в листинге 15.13 полностью эквивалентны.

Листинг 15.13. Интерполяция неявно вызывает метод `to_s`. Файл interpolation.rb

```
puts "Цена билета #{500} рублей"      # Цена билета 500 рублей
puts "Цена билета #{500.to_s} рублей" # Цена билета 500 рублей
```

Метод `to_s` поддерживается всеми объектами, поэтому интерполировать можно любой объект без опаски (листинг 15.14).

Листинг 15.14. Интерполяция объекта в строку. Файл ticket_inter.rb

```
require_relative 'ticket'

t = Ticket.new 500
puts "Объект билета: #{t}" # Объект билета: #<Ticket:0x00007fa8a40b65b8>
```

По умолчанию метод `to_s` объекта возвращает строку с именем класса и адресом объекта в оперативной памяти. В приведенном примере в качестве класса выступает `Ticket`, а в качестве адреса — `0x00007fa8a40b65b8`.

Такое представление может быть полезно при сравнении объектов. Однако зрителям гораздо важнее было бы знать о цене билета или дате проведения мероприятия — всех тех свойствах, которые имеют значение в реальном, а не в компьютерном мире.

Разработаем новую версию класса билета `Ticket`, объекты которой при интерполяции будут сообщать цену и дату проведения мероприятия. Для этого в классе необходимо реализовать метод `to_s`, который заменит стандартный метод (листинг 15.15).

Листинг 15.15. Перегрузка метода `to_s` в классе `Ticket`. Файл ticket_to_s.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
```

```
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end
end
```

В листинге 15.16 приводится пример использования класса.

Листинг 15.16. Интерполяция билета в строку. Файл ticket_to_s_use.rb

```
require_relative 'ticket_to_s'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)
puts "Билет: #{ticket}"
```

Результатом работы программы будет следующая строка:

```
Билет: цена 500, дата 2019-05-11 10:20:00 +0300
```

Однако сложить объект билета со строкой по-прежнему нельзя (листинг 15.17).

Листинг 15.17. Некорректное сложение строки с билетом. Файл ticket_to_s_sum.rb

```
require_relative 'ticket_to_s'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)
puts 'Билет: ' + ticket
```

При попытке запустить программу возникает следующая ошибка:

```
`+': no implicit conversion of Ticket into String (TypeError)
```

Можно было бы реализовать метод `+`, однако правильнее «превратить» объект `ticket` в строку. Для этого в класс `Ticket` следует добавить метод `to_str`. Именно наличие метода `to_str` позволяет складывать объекты со строками.

Метод `to_s` есть у любого объекта, а метод `to_str` — только у строк:

```
 > 'hello'.to_s
=> "hello"
 > 3.to_s
=> "3"
 > 'hello'.to_str
=> "hello"
 > 3.to_str
NoMethodError (undefined method `to_str' for 3:Integer)
```

На первый взгляд, метод `to_str` не выполняет никакой полезной работы — просто возвращает строку без изменения. Однако именно наличие метода `to_str` у объекта служит признаком строки. Поэтому метод `to_str` для строк необходим. Именно

наличие этого метода, а не принадлежность классу `String`, позволяет складывать строку друг с другом.

Если научить объект билета отзываться на `to_str`, для интерпретатора объект билета «превратится» в строку. В листинге 15.18 приводится пример обновленного класса `Ticket`, который реализует инстанс-методы `to_s` и `to_str`.

Листинг 15.18. Перегрузка метода `to_str` в классе `Ticket`. Файл ticket_to_str.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end

  alias to_str to_s
end
```

Для того, чтобы дословно не повторять содержимое методов `to_s` и `to_str`, для создания метода `to_str` здесь было использовано ключевое слово `alias`. Ключевое слово принимает в качестве первого аргумента новый метод, а в качестве второго — существующий.

Возможны и другие варианты создания метода `to_str` — например, вызов метода `to_s` из метода `to_str`:

```
def to_str
  to_s
end
```

Однако применение ключевого слова `alias` — это самый короткий вариант реализации метода `to_str`. В листинге 15.19 приводится пример использования класса `Ticket`. На этот раз сложение билета со строкой протекает корректно.

Листинг 15.19. Успешное сложение строки с билетом. Файл ticket_to_str_use.rb

```
require_relative 'ticket_to_str'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)
puts 'Билет: ' + ticket
# Билет: цена 500, дата 2019-05-11 10:20:00 +0300
```

15.6. Сложение объекта и массива

Для того, чтобы заставить объект вести себя как массив, не достаточно лишь одного метода (см. *разд. 21.4*). Тем не менее можно придать объекту некоторые свойства массива.

В объекте билета имеются два свойства: цена `price` и дата мероприятия `date`. Если пользователь захочет получить эти свойства в виде массива, он будет искать стандартный метод `to_a`. В листинге 15.20 приводится пример класса `Ticket` с реализацией этого метода.

Листинг 15.20. Реализация метода `to_a`. Файл ticket_to_a.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end

  alias to_str to_s

  def to_a
    [price, date]
  end
end
```

Метод `to_a` в объекте билета позволяет разработчику в любой момент получить данные объекта в виде массива. Кроме того, метод станет неявно вызываться всякий раз, когда билет будет оказываться в контексте массива, — например, при передаче объекта методу `Array` (листинг 15.21).

Листинг 15.21. Использование метода `to_a`. Файл ticket_to_a_use.rb

```
require_relative 'ticket_to_a'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)

p ticket.to_a   # [500, 2019-05-11 10:20:00 +0300]
p Array(ticket) # [500, 2019-05-11 10:20:00 +0300]
```

Однако попытка сложения билета с массивом завершается неудачей (листинг 15.22).

Листинг 15.22. Некорректное сложение билета с массивом. Файл ticket_to_a_sum.rb

```
require_relative 'ticket_to_a'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)

p [3, 5] + ticket
```

Программа завершается сообщением об ошибке:

```
no implicit conversion of Ticket into Array (TypeError)
```

Поправить ситуацию можно явным вызовом метода `to_a` при сложении массива и объекта:

```
p [3, 5] + ticket.to_a # [3, 5, 500, 2019-05-11 10:20:00 +0300]
```

Однако, если необходимо, чтобы при сложении объект вел себя как массив, нужно переопределить метод `to_ary` (листинг 15.23).

Листинг 15.23. Реализация метода `to_ary`. Файл ticket_to_ary.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end

  def to_a
    [price, date]
  end

  alias to_str to_s
  alias to_ary to_a
end
```

Метод `to_ary` делает то же самое, что и метод `to_a`. Поэтому нет необходимости в его полноценной реализации. Достаточно ввести псевдоним при помощи ключевого слова `alias`. Теперь объект билета можно складывать с массивом без явного преобразования (листинг 15.24).

Листинг 15.24. Корректное сложение массива с билетом. Файл ticket_to_ary_use.rb

```
require_relative 'ticket_to_ary'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)
p [3, 5] + ticket # [3, 5, 500, 2019-05-11 10:20:00 +0300]
```

Объекты массива сами содержат методы `to_a` и `to_ary`. На первый взгляд может показаться, что эти методы совершенно бесполезны, т. к. никак не изменяют объект:

```
 > [1, 2].to_a
=> [1, 2]
 > [1, 2].to_ary
=> [1, 2]
```

Методы `to_a` и `to_arr` у объектов массива выступают признаками массива. Коллекционные объекты весьма сложны, не достаточно лишь этих двух методов, чтобы объект считался массивом.

15.7. Перегрузка *[]* и *[]=*

В строках и массивах для доступа к отдельным элементам можно использовать квадратные скобки. Для этого внутри квадратных скобок размещается индекс элемента, который отсчитывается с нуля:

```
 > s = 'Hello, world!'
=> "Hello, world!"
 > s[4]
=> "o"
 > a = [*1..5]
=> [1, 2, 3, 4, 5]
 > a[2]
=> 3
```

Переопределение поведения оператора — например, квадратных скобок: `ticket[0]`, называется *перегрузкой оператора*.

Для поддержки квадратных скобок в объекте необходимо реализовать два метода: `[]` — для доступа к элементам и `[]=` — для назначения нового значения (листинг 15.25).

Листинг 15.25. Перегрузка квадратных скобок. Файл ticket_bracket_overload.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
```

```
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end

  def to_a
    [price, date]
  end

  def [](index)
    to_a[index]
  end

  def []=(index, value)
    case index
    when 0 then @price = value
    when 1 then @date = value
    end
  end

  alias to_str to_s
  alias to_ary to_a
end
```

Метод `[]` принимает единственный числовой параметр `index`. Массив в объекте билета содержит только два элемента: `[price, date]` (цену и дату). Поэтому в качестве аргумента метода ожидаются цифры `0` и `1`. В случае любых других значений будет возвращаться неопределенное значение `nil`.

Внутри метода `[]` индекс `index` передается массиву, который формируется методом `to_a`. В случае присваивания воспользоваться массивом уже не получится, т. к. массив `to_a` предназначен только для чтения.

Метод `[]=` предназначен для программирования операций вида `ticket[0] = 600`. Чтобы совершить такую операцию, необходимы два значения: индекс в квадратных скобках и значение после оператора присваивания. Поэтому метод принимает два параметра: `index` (индекс) и `value` (значение).

Допустимых значений индекса всего два — это не так много. Поэтому для выбора инстанс-переменной `@price` или `@date` используется `case`-выражение.

В листинге 15.26 приводится пример использования класса `Ticket` с перегруженными квадратными скобками.

Листинг 15.26. Файл ticket_bracket_overload_use.rb

```
require_relative 'ticket_bracket_overload'

ticket = Ticket.new date: Time.mktime(2019, 5, 10, 10, 20)
```

```
p ticket[0] # 500
p ticket[1] # 2019-05-10 10:20:00 +0300

ticket[0] = 600
ticket[1] = Time.mktime(2019, 5, 11, 10, 20)

p ticket[0] # 600
p ticket[1] # 2019-05-11 10:20:00 +0300
```

Способ доступа к элементам получился не очень удобным. Необходимо всегда помнить, что индекс `0` — это цена, а `1` — дата и время мероприятия. Было бы гораздо удобнее, если бы вместо числовых индексов использовались строковые или символьные ключи.

Такой подход не только повысит наглядность программы, но и позволит избавиться от `case`-выражения в методе `[]=`. В листинге 15.27 приводится пример обновленного класса `Ticket`. Поведение квадратных скобок в нем больше напоминает хэш, нежели массив.

Листинг 15.27. Добавление поведения хэша. Файл ticket_hash.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
    @price = price
  end

  def to_s
    "цена #{price}, дата #{date}"
  end

  def to_a
    [price, date]
  end

  def [](key)
    send(key) if respond_to? key
  end

  def []=(key, *value)
    method = "#{key}=".to_sym
    send(method, *value) if respond_to? method
  end

  alias to_str to_s
  alias to_ary to_a
end
```

Метод `[]` здесь принимает в качестве аргумента символ `key` и передает его без изменения методу `send` (см. *разд. 14.6*). Метод `send` вызывается только в том случае, если объект билета поддерживает метод с именем `key`. То есть передача в качестве параметра `key` значений `:price` и `:date` приведет к вызову геттеров цены и даты. Если в `key` передать значение несуществующего геттера, такой вызов будет проигнорирован.

В методе `[]=` необходимо преобразовать символы `:price` и `:date` в `:price=` и `:date=` соответственно. Это позволит обратиться к методам присваивания цены и даты `price=` и `date=`, которые создает метод `attr_accessor`.

В листинге 15.28 приводится пример использования обновленного класса `Ticket`.

Листинг 15.28. Файл ticket_hash_use.rb

```
require_relative 'ticket_hash'

ticket = Ticket.new date: Time.mktime(2019, 5, 10, 10, 20)

p ticket[:price] # 500
p ticket[:date]  # 2019-05-10 10:20:00 +0300

ticket[:price] = 600
ticket[:date]  = Time.mktime(2019, 5, 11, 10, 20)

p ticket[:price] # 600
p ticket[:date]  # 2019-05-11 10:20:00 +0300
```

15.8. Перегрузка унарных операторов +, – и !

Количество операндов, которые необходимы для оператора, может различаться. Это могут быть один, два или три операнда. При этом говорят об *унарном*, *бинарном* и *тернарном* операторах.

Тернарный оператор в Ruby один — это логический оператор `x ? y : z`:

```
 > RUBY_VERSION == '2.5.3' ? 'Корректная версия' : 'Некорректная версия'
=> "Некорректная версия"
```

Большинство операторов в Ruby бинарные:

```
 > 5 + 2
=> 7
 > 'hello' == 'h' + 'e' + 'l' * 2 + 'o'
=> true
```

Однако имеются и три унарных оператора: знак «плюс» `+`, знак «минус» `-` и логическое отрицание `!`:

```
 > +10
=> 10
 > -10
=> -10
 > !nil
=> true
```

Внешне унарный + ничем не отличается от бинарного. Интерпретатор-Ruby «понимает» из контекста, что в выражении 5 + 2 — бинарный «плюс», а не унарный, как здесь: +2. Однако при перегрузке операторов это сделать невозможно. Поэтому по умолчанию считается, что перегружается бинарный вариант оператора (см. листинг 15.10).

При перегрузке унарных операторов действуют особые правила — «плюс» обозначается символом: +@, «минус»: -@, а логическое отрицание: !@ (листинг 15.29).

Листинг 15.29. Перегрузка унарных операторов Файл unary.rb

```
class HelloWorld
  def +@
    puts 'Перегрузка операции +hello'
  end

  def -@
    puts 'Перегрузка операции -hello'
  end

  def !@
    puts 'Перегрузка операции !hello'
    true
  end
end

hello = HelloWorld.new

+hello # Перегрузка операции +hello
-hello # Перегрузка операции -hello
!hello # Перегрузка операции !hello
```

15.9. Какие операторы можно перегружать?

Как видно из предыдущих разделов, допускается перегружать практически все операторы языка Ruby. Тем не менее нельзя взять совершенно произвольную последовательность символов, которой нет в языке, и определить для нее поведение. Попытка определить в классе Ticket произвольный оператор <=/=> завершится сообщением об ошибке (листинг 15.30).

Листинг 15.30. Не все операторы можно перегружать. Файл any_operator.rb

```
class Ticket
  def <=/=>(obj)
    # TODO
  end
end

ticket = Ticket.new
```

Список операторов, для которых допускается перегрузка, приводится в табл. 15.2.

Таблица 15.2. *Список операторов, для которых допускается перегрузка*

Оператор	Описание	Пример
+	Арифметический «плюс»	5 + 2
-	Арифметический «минус»	5 - 2
*	Умножение	2 * 3
**	Возведение в степень	2 ** 3
/	Деление	8 / 2
%	Остаток от деления	27 % 5
&	Поразрядное И	5 & 4
|	Поразрядное ИЛИ	5 | 4
^	Поразрядное исключающее ИЛИ	5 ^ 4
>>	Правый поразрядный сдвиг	5 >> 2
<<	Левый поразрядный сдвиг	5 << 2
==	Логическое равенство	2 == '2'.to_i
===	Оператор сравнения в case	Integer === 3
=~	Соответствие регулярному выражению	// =~ ''
!~	Несоответствие регулярному выражению	// !~ ''
<=>	Оператор сравнения	5 <=> 2
<	Оператор «меньше»	5 < 2
<=	Оператор «меньше равно»	5 <= 2
>	Оператор «больше»	5 > 2
>=	Оператор «больше равно»	5 >= 2
+	Унарный «плюс»	+5
-	Унарный «минус»	-5
!	Логическое отрицание	!5.nil?
[]	Квадратные скобки	hello[1]
[]=	Квадратные скобки с присваиванием	hello[1] = 'a'

15.10. DuckType-типизация

Объект в Ruby считается строкой, массивом, хэшем не из-за принадлежности классу `String`, `Array` или `Hash`. Поведение объекта определяется поддержкой операторов и методов, которые являются признаками строки, массива или хэша. Любой объект или класс может «притвориться» любым другим объектом или классом.

В Ruby нет строгих типов, как в языках программирования Java, C++ или PHP. Языки программирования, которые в отношении типов ведут себя как Ruby, называют языками с *утиной типизацией* (Duck Typing). Помимо Ruby, к ним относятся такие языки программирования, как Smalltalk или Python.

Согласно утиной типизации, если объект ходит как утка, плавает как утка, крякает как утка — это и есть утка. Применительно к Ruby это означает, что если объект отзывается на метод `to_s` и `to_str`, то этот объект является строкой. При этом класс объекта не обязан быть классом `String`.

Именно поэтому ориентироваться на имена классов в Ruby крайне опасно. Можно открыть любой класс и расширить его возможности или, наоборот, удалить из него какие-либо методы. Можно создать объект класса и изменить его при помощи синглетон-методов. Рассмотрим особенности утиной типизации в Ruby на примерах.

Узнать название класса можно при помощи метода `class`. Если имеются какие-то сомнения, можно в любой момент поэкспериментировать с объектом в консоли интерактивного Ruby (`irb`):

```
 > 'hello'.class
=> String
 > 3.class
=> Integer
```

Более того, каждый объект предоставляет метод `instance_of?`, который принимает в качестве аргумента класс и возвращает «истину» `true`, если текущий объект — это инстанс класса, иначе возвращается «ложь» `false`:

```
 > 'hello'.instance_of? String
=> true
 > 'hello'.instance_of? Integer
=> false
```

Однако принадлежность к классу еще не означает, что объект всегда будет вести себя как строка или, наоборот, объект другого класса не может вести себя как строковый объект. Методы `to_s` и `to_str` можно добавить в класс объекта, а из класса `String`, наоборот, удалить поддержку этих методов. В листинге 15.31 создаются два объекта: `ticket` и `hello`. В классе `Ticket` определены два метода: `to_s` и `to_str`, класс строки `String` открывается, и из него при помощи ключевого слова `undef` удаляются методы `to_s` и `to_str`.

Листинг 15.31. Использование метода `instance_of?`. Файл instance_of.rb

```
require_relative 'ticket_to_str'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)

print 'Объект ticket '
if ticket.instance_of?(String)
  puts 'является строкой'
else
  puts 'не является строкой'
end

hello = 'hello'

class String
  undef to_s
  undef to_str
end

print 'Объект hello '
if hello.instance_of?(String)
  puts 'является строкой'
else
  puts 'не является строкой'
end

puts hello.to_s
```

Результатом выполнения программы будут следующие строки:

```
$ ruby instance_of.rb
Объект ticket не является строкой
Объект hello является строкой
undefined method `to_s' for "hello":String (NoMethodError)
```

Несмотря на то, что объект `ticket` отзывается на методы `to_s` и `to_str`, его класс `Ticket` не позволяет здесь рассматривать его как строку. Удаление методов `to_s` и `to_str` из класса `String` не изменяет его класс, и объект `hello` по-прежнему рассматривается как строка. При этом попытка вызова метода `to_s` в отношении объекта `hello` завершается ошибкой.

Метод `instance_of?` — это инструмент типизированных языков, в языках, поддерживающих утиную типизацию, более востребованным оказывается метод `respond_to?`, который проверяет, отзывается ли объект на метод (листинг 15.32).

Листинг 15.32. Использование метода `respond_to?`. Файл respond_to.rb

```
require_relative 'ticket_to_str'

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)
```

```
print 'Объект ticket '
if ticket.respond_to?(:to_s) && ticket.respond_to?(:to_str)
  puts 'является строкой'
else
  puts 'не является строкой'
end

hello = 'hello'

class String
  undef to_s
  undef to_str
end

print 'Объект hello '
if hello.respond_to?(:to_s) && hello.respond_to?(:to_str)
  puts 'является строкой'
else
  puts 'не является строкой'
end
```

Результатом выполнения программы будет противоположный результат — билет теперь является строкой, а бывшая строка — нет:

```
Объект ticket является строкой
Объект hello не является строкой
```

В предопределенных классах Ruby также имеются примеры, когда объекты реализуют интерфейсы других классов. Так, предопределенная константа `ENV` представляет переменные окружения. При помощи этого объекта можно получить доступ ко всем переменным окружения, заданным в командной оболочке:

```
 > ENV
=> {"NVM_DIR"=>"/Users/i.simdyanov/.nvm", "_system_type"=>"Darwin",
"LANG"=>"ru_RU.UTF-8", ..., "LINES"=>"45", "COLUMNS"=>"202"}
 > ENV['LANG']
=> "ru_RU.UTF-8"
```

В объект можно даже добавить свои собственные значения, как в объект класса `Hash`:

```
 > ENV['hello'] = 'world'
=> "world"
 > ENV
=> {"NVM_DIR"=>"/Users/i.simdyanov/.nvm", "_system_type"=>"Darwin",
"LANG"=>"ru_RU.UTF-8", ..., "LINES"=>"45", "COLUMNS"=>"202", "hello"=>"world"}
```

Тем не менее `ENV` — это объект класса `Object`, который лишь «притворяется» хэшем:

```
 > ENV.class
=> Object
```

Задания

1. Преобразуйте число 100 в двоичное представление 1100100, восьмеричное — 144 и шестнадцатеричное — 64.
2. Дан массив:

   ```
   %i(first second third)
   ```

   Превратите его в хэш:

   ```
   {first: 1, second: 2, third: 3}
   ```

3. Преобразуйте массив вида:

   ```
   %w[first second third]
   ```

   в хэш вида:

   ```
   {first: 'Fst (1)', second: 'Snd (2)', third: 'Trd (3)'}
   ```

   Выражение преобразования должно быть однострочным.

4. Возьмите текст этого задания и извлеките из него все слова, количество символов в которых больше 5. Подсчитайте количество слов и выведите их в алфавитном порядке.
5. Возьмите текст этого задания и извлеките из него все слова, которые начинаются с символа `'и'`. Сформируйте из них список уникальных слов и выведите их в порядке увеличения количества символов в слове.
6. Имеются два массива:

   ```
   %w[red orange yellow green gray indigo violet]
   ```

   и

   ```
   %w[красный оранжевый желтый зеленый голубой синий фиолетовый]
   ```

   Создайте из них хэш:

   ```
   {red: 'красный', orange: 'оранжевый', yellow: 'желтый', green: 'зеленый',
   blue: 'голубой', indigo: 'синий', violet: 'фиолетовый'}
   ```

ГЛАВА 16

Ключевое слово *self*

Файлы с исходными кодами этой главы находятся в каталоге *self* сопровождающего книгу электронного архива.

Объекты — это весьма сложные структуры, расположенные в оперативной памяти компьютера. При операциях с ними: присваивании переменным, передаче в качестве аргумента или назначении в качестве элемента массива — не происходит перемещения объекта. Вместо этого на объект создается *ссылка*. Ссылка занимает гораздо меньше памяти, поэтому перемещается значительно быстрее. Так что, вместо физического перемещения объектов просто перемещаются ссылки на объект.

Получить доступ к объекту можно по имени переменной. Однако иногда переменной недостаточно. Язык программирования Ruby предоставляет специальное ключевое слово `self`, которое ссылается на текущий объект.

В предыдущих главах слово `self` уже неоднократно использовалось. В этой главе мы систематизируем все сведения об этом ключевом слове.

16.1. Ссылки на текущий объект

Ключевое слово `self` — это ссылка на текущий объект. В *главе 2* упоминалось, что у методов всегда есть получатель. В том случае, если он опускается, подразумевается, что в качестве получателя выступает ключевое слово `self` (листинг 16.1).

Листинг 16.1. Использование ключевого слова `self`. Файл ticket.rb

```
class Ticket
  def initialize(price: 500)
    @price = price
  end

  def price
    @price
  end
```

```
  def to_s
    "цена: #{self.price}"
  end
end

ticket = Ticket.new(price: 600)

puts ticket.to_s
```

В листинге 16.1 создается класс билета `Ticket`, в котором используется единственная инстанс-переменная для цены: `@price`. Для доступа к переменной создается метод-геттер `price`. В методе `to_s` метод `price` вызывается, причем в качестве получателя используется ключевое слово `self`.

В этом случае использование `self` избыточно — если получатель не указан, подразумевается, что им выступает текущий объект. Именно поэтому до текущего момента `self` почти никогда не вызывалось явно. Считается хорошим тоном опускать его везде, где это возможно.

Значение `self` не является постоянным — внутри инстанс- и синглетон-методов в теле класса и в глобальной области ссылка на текущий объект постоянно изменяется. Так как `self` при вызове методов — получатель неявный, крайне важно понимать, на что ссылается это ключевое слово в каждый момент времени.

Иногда обойтись без ключевого слова `self` невозможно. Рассмотрим ситуацию на примере инициализации инстанс-переменных в методе `initialize`. Для этого воспользуемся классом `Ticket` (листинг 16.2).

Листинг 16.2. Инициализация инстанс-переменных. Файл initialize.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    @date = date
    @price = price
  end
end

p Ticket.instance_methods(false) # [:date, :date=, :price, :price=]
```

Здесь с помощью метода `attr_accessor` вводятся по две пары методов для каждой из инстанс-переменных: `price` и `price=` — для цены и `date` и `date=` — для даты.

Получить список созданных в классе методов удобно при помощи метода `instance_methods`. Он очень похож на метод `methods`, только, в отличие от него, возвращает не список методов для вызова, а список инстанс-методов, т. е. методов, которые будут доступны в объекте этого класса. По умолчанию метод возвращает их полный список, включая предопределенные методы. Однако, если ему передать

необязательный аргумент `false`, в списке будут содержаться только те методы, которые определены в классе `Ticket`.

Как результат, программа выводит массив из четырех символов:

```
[:date, :date=, :price, :price=]
```

Это методы, которые были добавлены `attr_accessor`-ом: два геттера и два сеттера. Внутри метода `initialize` вместо явного присваивания значений инстанс-переменным можно было бы воспользоваться сеттерами (листинг 16.3).

Листинг 16.3. Некорректное использование сеттеров. Файл initialize_wrong.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    date = date
    price = price
  end
end

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)

p ticket.price # nil
p ticket.date  # nil
```

Как видно из результатов, цена и дата остаются неинициализированными — интерпретатор Ruby не воспринимает `date =` и `price =` как методы. Вместо этого выражения рассматриваются как присваивание локальным переменным `price` и `date`.

Для того чтобы воспользоваться методами-сеттерами, необходимо явно указать получатель. Здесь в качестве получателя выступает ключевое слово `self` (листинг 16.4).

Листинг 16.4. Корректное использование сеттеров. Файл initialize_self.rb

```
class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    self.date = date
    self.price = price
  end
end

ticket = Ticket.new date: Time.mktime(2019, 5, 11, 10, 20)

p ticket.price # 500
p ticket.date  # 2019-05-11 10:20:00 +0300
```

Как видно из листинга 16.4, инстанс-переменные успешно инициализируются. Во всех других случаях `self` никогда не указывается явно. Если Ruby допускает не указывать явно какую-либо синтаксическую конструкцию, Ruby-разработчики никогда ее не указывают.

16.2. Значения *self* в разных контекстах

Ключевое слово `self` в разных точках программы ссылается на разные объекты. Всего различают несколько зон:

- глобальная область;
- инстанс-метод;
- уровень класса или модуля;
- синглетон-метод.

Переход через границы указанных областей приводит к тому, что `self` начинает ссылаться на другой объект.

В глобальной области `self` ссылается на объект `main`, который имеет класс `Object` (листинг 16.5).

Листинг 16.5. `self` в глобальной области. Файл main.rb

```
p self          # main
puts self.class # Object
```

Строку `'main'` объект возвращает за счет перегруженного метода `to_s`. Явный вызов `self.to_s` приводит к аналогичному результату. Объект глобальной области не имеет переменной по умолчанию, единственный способ получить на него ссылку — это воспользоваться `self`.

В классах и модулях ключевое слово `self` ссылается на сам класс или модуль (листинг 16.6).

Листинг 16.6. `self` в классе. Файл class.rb

```
class Ticket
  puts self   # Ticket
  module Summary
    puts self # Ticket::Summary
  end
  puts self   # Ticket
end
```

Инстанс-методы — это методы, которые будут доступны объекту класса. На момент определения инстанс-метода объект класса пока не существует. Однако на него уже можно сослаться при помощи ключевого слова `self` (листинг 16.7).

Листинг 16.7. `self` в инстанс-методе. Файл instance_method.rb

```
class Ticket
  def price
    puts self
  end
end

ticket = Ticket.new
ticket.price # #<Ticket:0x007fd57c8276d0>
p ticket     # #<Ticket:0x007ff592012630>
```

Чтобы воспользоваться методом `price`, необходимо создать объект `ticket`. Как можно видеть, `self` ссылается на объект `ticket`.

На уровне синглетон-метода ключевое слово `self` ссылается на объект получателя. В листинге 16.8 создается объект `ticket`, в который добавляется синглетон-метод `price`. Метод в качестве результата возвращает значение `self`.

Листинг 16.8. `self` в синглетон-методе. Файл singleton_method.rb

```
ticket = Object.new

def ticket.price
  self
end

p ticket.price # #<Object:0x007fee1b8129f8>
p ticket       # #<Object:0x007fee1b8129f8>
```

16.3. Приемы использования *self*

Из поведения ключевого слова `self` в разных контекстах имеется несколько следствий.

16.3.1. Методы класса

Ключевое слово `self` можно использовать при определении методов класса, которые по своей природе являются синглетон-методами. В листинге 16.9 представлен класс `Ticket`, для которого определяется метод `Ticket.max_count`, возвращающий максимально допустимое количество билетов. Это количество задается при помощи константы `MAX_COUNT`.

Листинг 16.9. Метод `max_count`. Файл max_count_singleton.rb

```
class Ticket
  MAX_COUNT = 300
end
```

```
def Ticket.max_count
  Ticket::MAX_COUNT
end

puts Ticket.max_count # 300
```

Синглетон-метод можно разместить в теле метода класса и при обращении к константе избавиться от префикса `Ticket::`. Внутри тела класса `self` ссылается на объект `Ticket`. Это две ссылки на один и тот же объект класса, поэтому они взаимозаменяемы. В листинге 16.10 в качестве получателя метода `max_count` вместо класса `Ticket` выступает ключевое слово `self`.

Листинг 16.10. Метод `max_count`. Файл max_count_self.rb

```
class Ticket
  MAX_COUNT = 300

  def self.max_count # Ticket.max_count
    MAX_COUNT
  end
end

puts Ticket.max_count # 300
```

В обращении к константе `Ticket::MAX_COUNT` имя класса тоже можно было заменить ключевым словом `self`:

```
def self.max_count
  self::MAX_COUNT
end
```

Однако прямой вызов константы без префикса `self::` выглядит короче и более читаемо. Поэтому на практике `self::` перед константами всегда опускается.

В теле класса может быть расположено несколько методов класса (листинг 16.11).

Листинг 16.11. Несколько методов класса. Файл class_methods.rb

```
class Ticket
  MAX_COUNT = 300
  MAX_PRICE = 1200

  def self.max_count
    MAX_COUNT
  end

  def self.max_price
    MAX_PRICE
  end
end
```

```
puts Ticket.max_count # 300
puts Ticket.max_price # 1200
```

Расширить класс `Ticket` синглетон-методами можно при помощи ключевого слова `class` и оператора `<<` (листинг 16.12).

Листинг 16.12. Альтернативный способ. Файл class_methods_class.rb

```
class Ticket
  MAX_COUNT = 300
  MAX_PRICE = 1200
end

class << Ticket
  def max_count
    Ticket::MAX_COUNT
  end

  def max_price
    Ticket::MAX_PRICE
  end
end

puts Ticket.max_count # 300
puts Ticket.max_price # 1200
```

Конструкцию `class` можно разместить внутри тела класса. Константу `Ticket` после `<<` можно заменить на `self`. Перед константами `MAX_COUNT` и `MAX_PRICE` префикс `Ticket::` можно вообще убрать (листинг 16.13).

Листинг 16.13. Файл class_methods_alter.rb

```
class Ticket
  MAX_COUNT = 300
  MAX_PRICE = 1200

  class << self
    def max_count
      MAX_COUNT
    end

    def max_price
      MAX_PRICE
    end
  end
end

puts Ticket.max_count # 300
puts Ticket.max_price # 1200
```

Способы определения методов класса из листингов 16.11 и 16.13 являются одними из самых популярных в Ruby-сообществе. Как правило, к варианту `self.max_count` прибегают, когда нужен один метод. Конструкцию `class <<` используют, когда необходимо определить сразу несколько методов класса.

16.3.2. Цепочка обязанностей

В Ruby методы можно вызывать друг за другом, объединяя их при помощи оператора «точка» в цепочку:

```
 > 'hello'.capitalize
=> "Hello"
 > 'hello'.capitalize.reverse
=> "olleH"
```

В приведенном примере с помощью метода `capitalize` первая буква строки сначала превращается в заглавную, а затем с помощью метода `reverse` порядок следования символов в строке меняется на обратный.

Чтобы самостоятельно реализовать методы, способные участвовать в такой цепочке, методы всякий раз должны возвращать объект. Это удобно делать, возвращая в качестве результата `self`.

В листинге 16.14 приводится класс `Ticket` с двумя инстанс-переменными: `@price` (цена) и `@status` (статус). Если статус равен `true`, билет доступен для продажи, если `false` — билет куплен.

Листинг 16.14. Цепочка обязанностей. Файл chain.rb

```
class Ticket
  attr_accessor :price, :status

  def initialize(price:)
    @price = price
    @status = true
  end

  def buy
    @status = false
    self
  end
end
```

Здесь при создании объекта в методе `initialize` инстанс-переменная `@status` получает значение `true`. Кроме того, в классе реализован метод `buy`, вызов которого резервирует текущий объект билета. В результате вызова метода инстанс-переменная `@status` получает значение `false`. В конце метода `buy` возвращается ссылка на текущий объект `self`. Это позволяет сразу после метода `buy` вызывать другие методы этого объекта (листинг 16.15).

Листинг 16.15. Использование цепочки обязанностей. Файл chain_use.rb

```
require_relative 'chain'

ticket = Ticket.new price: 600
puts ticket.buy.price # 600
```

В приведенном примере в одной строке происходит покупка билета методом `buy` и тут же возвращается цена билета методом `price`.

16.3.3. Перегрузка операторов

Пусть имеется класс `Ticket` с ценой `@price` (см. листинг 16.14), и нам требуется при сложении билета с числом увеличивать цену билета на это число. Для этого необходимо перегрузить оператор `+` (листинг 16.16).

Листинг 16.16. Перегрузка оператора +. Файл plus_overload.rb

```
class Ticket
  attr_accessor :price

  def initialize(price:)
    @price = price
  end

  def +(number)
    @price += number
    self
  end
end
```

В листинге 16.17 приводится пример сложения билета с числом `100`. В результате цена билета увеличивается с `500` до `600`.

Листинг 16.17. Использование перегруженного оператора. Файл plus_overload_use.rb

```
require_relative 'plus_overload'

ticket = Ticket.new(price: 500)

ticket = ticket + 100
puts ticket.price # 600
```

Просто прибавить к инстанс-переменной число недостаточно, т. к. в этом случае оператор `+` будет возвращать число. В конце метода `+` необходимо вернуть ссылку на текущий объект. Если этого не сделать, последним выражением метода будет

`@price += number`, которое возвращает число `600`, а не объект класса `Ticket`. Разумеется, обращение `600.price` будет завершаться ошибкой.

16.3.4. Инициализация объекта блоком

Ссылку на текущий объект можно передавать в блок. Для этого `self` следует передать в качестве аргумента ключевому слову `yield`. В листинге 16.18 приводится пример инициализации объекта блоком.

Листинг 16.18. Инициализация объекта блоком. Файл initialize_yield_self.rb

```
class Ticket
  attr_accessor :date, :price

  def initialize
    yield self
  end
end

ticket = Ticket.new do |t|
  t.price = 600
  t.date = Time.mktime(2019, 5, 11, 10, 20)
end

p ticket.price # 600
p ticket.date  # 2019-05-11 10:20:00 +0300
```

Инстанс-переменные `@date` и `@price` можно инициализировать в блоке. Так как объект передается по ссылке, все изменения, которые производятся внутри блока, будут отражаться на состоянии объекта.

16.3.5. Открытие класса

Ключевое слово `self` интенсивно используется при открытии классов. В листинге 16.19 приводится пример открытия стандартного класса `String` и добавления в него метода `hello`, который возвращает строку приветствия. Для доступа к текущей строке используется `self`.

Листинг 16.19. Открытие класса `String`. Файл string.rb

```
class String
  def hello
    "Hello, #{self}!"
  end
end

p 'Ruby'.hello # Hello, Ruby!
```

В результате каждая строка получает метод `hello`, обращение к которому позволяет вывести приветствие с участием текущей строки.

В листинге 16.20 открывается класс `Integer`, в который добавляются методы `minutes`, `hours` и `days`. Каждый из этих методов возвращает количество секунд — например, вызов `1.minutes` возвращает 60 секунд, `2.minutes` — 120 секунд и т. д.

Листинг 16.20. Открытие класса `Integer`. Файл integer.rb

```
class Integer
  SEC_PER_MINUTE = 60
  SEC_PER_HOUR = 3_600
  SEC_PER_DAY = 86_400

  def minutes
    self * SEC_PER_MINUTE
  end

  def hours
    self * SEC_PER_HOUR
  end

  def days
    self * SEC_PER_DAY
  end
end

puts 10.minutes # 600
puts 5.hours    # 18000
puts 2.days     # 172800
```

Веб-фреймворк Ruby on Rails аналогичным образом расширяет классы языка Ruby, добавляя помимо методов `minutes`, `hours` и `days` множество других, — например, для пересчета кило-, мега- и гигабайтов в байты.

Задания

1. Откройте класс `Integer` и добавьте в него методы `kilobytes`, `megabytes`, `gigabytes` и `terabytes`. Вызов `5.kilobytes` должен вернуть количество байтов в пяти килобайтах — 5120. Аналогично, для остальных методов необходимо вычислить байты в соответствующем количестве мегабайт, гигабайт и терабайт.
2. Создайте класс пользователя `User`, в объекте которого можно сохранить фамилию, имя, отчество пользователя, а также его электронный адрес. Добейтесь, чтобы метод `new` класса `User` принимал блок, в котором можно инициализировать объект.

ГЛАВА 17

Наследование

Файлы с исходными кодами этой главы находятся в каталоге *inheritance* сопровождающего книгу электронного архива.

Одной из главных целей объектно-ориентированного подхода является повторное использование кода. Достигается эта цель множеством средств. Однако ключевым среди них является *механизм наследования*, который описывается в этой главе.

17.1. Наследование

Классы можно связать отношением наследования, в котором один класс наследует методы и свойства другого класса. Класс-родитель называется *базовым классом*. Класс-потомок называется *производным классом*.

Предположим, в программе моделируется сложная классификация — например, весь доступный людям транспорт (рис. 17.1).

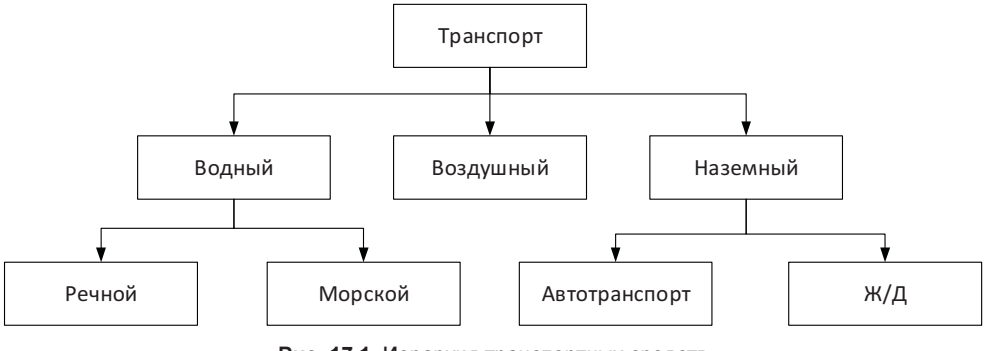


Рис. 17.1. Иерархия транспортных средств

Абстрактное понятие транспорта может дробиться на водный, воздушный, наземный. Далее каждую из категорий можно разбить на подкатегории. Дробление и уточнение можно продолжить вплоть до конкретных моделей транспортных средств.

У всех транспортных средств есть общие свойства:

- грузоподъемность;
- возможность перемещаться из одной точки в другую;
- скорость движения.

Эти обобщенные свойства, которые присущи всем транспортным средствам, можно было бы моделировать на уровне базового класса `Транспорт`. В производных классах `Водный`, `Воздушный` или `Наземный` можно было бы уточнять особенности движения в той или иной среде. Например, для водного транспорта будет иметь значение водоизмещение и время автономного плавания. Весь воздушный транспорт имеет предельный потолок по высоте. Для автотранспорта важное значение отводится виду топлива, а для железнодорожного — типу тока.

Рассмотрим инструменты наследования Ruby на более простом примере — например, на классах, которые могли бы использоваться для создания веб-сайта (рис. 17.2).

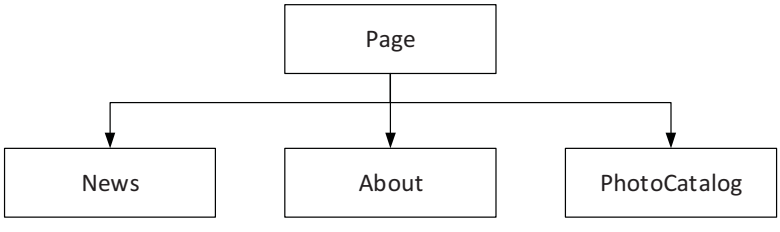


Рис. 17.2. Классы веб-сайта

Пусть веб-сайт содержит несколько типов страниц: новости — `News`, «О нас» — `About` и страница с фотогалереей — `PhotoCatalog`.

У всех страниц есть общие свойства, например: заголовок `title`, содержимое страницы `body` и ключевые слова `keywords`. В листинге 17.1 приводится возможная реализация класса `Page`, от которого будут наследоваться все остальные страницы.

Листинг 17.1. Базовый класс страницы `Page`. Файл page.rb

```
class Page
  attr_accessor :title, :body, :keywords
end
```

Теперь можно приступить к реализации базовых страниц. В листинге 17.2 приводится класс новостей `News`, который унаследован от класса `Page`.

Листинг 17.2. Производный класс новостной страницы `News`. Файл news.rb

```
require_relative 'page'

class News < Page
  attr_accessor :date
end
```

Наследование класса задается при помощи оператора <, после которого указывается базовый класс. Наследовать свойства и методы можно только от одного класса. Ruby не поддерживает *множественное наследование*.

Класс `News` уже содержит свойства `title`, `body` и `keywords`, наследуя их от класса `Page`. Кроме того, класс вводит дополнительное свойство `date` — дату и время публикации материала. Последний параметр важен для новостных материалов — чем более свежая новость, тем она ценнее.

Свойства страницы «О нас» задаются классом `About`, который также наследуется от класса `Page` (листинг 17.3). Расширим свойства класса `About` списком телефонов `phones` и адресом `address`.

Листинг 17.3. Производный класс страницы «О нас» `About`. Файл about.rb

```
require_relative 'page'

class About < Page
  attr_accessor :phones, :address
end
```

Остается реализовать последний класс: фотогалереи `PhotoCatalog`. От базового класса `Page` он отличается только списком фотографий `photos` (листинг 17.4).

Листинг 17.4. Производный класс фотогалереи `PhotoCatalog`. Файл photo_catalog.rb

```
require_relative 'page'

class PhotoCatalog < Page
  attr_accessor :photos
end
```

Теперь можно создать объект какой-либо страницы — например «О нас». Для быстрого подключения классов создадим массив файлов и подключим их при помощи директивы `require_relative` в итераторе `each` (листинг 17.5).

Листинг 17.5. Создание объекта класса `About`. Файл about_use.rb

```
%w[about news photo_catalog].each do |file|
  require_relative file
end

about = About.new

about.title = 'О нас'
about.body = 'Вы можете обнаружить нас по адресу'
about.keywords = ['О нас', 'Адреса', 'Телефоны']
about.phones = ['+7 920 4567722', '+7 920 4567733']

p about
```

Несмотря на то, что сеттеры `title=`, `body=` и `keywords=` реализованы в классе `Page`, они доступны и для объектов класса `About`. Результатом выполнения программы из листинга 17.5 будет следующая строка:

```
#<About:0x00007f8b930f1cb8 @title="О нас", @body="Вы можете обнаружить нас
по адресу", @keywords=["О нас", "Адреса", "Телефоны"], @phones=["+7 920 4567722",
"+7 920 4567733"]>
```

В листинге 17.6 создается объект класса `Page`. Это самый «бедный» по функциональности класс, воспользоваться в нем сеттерами производных классов не получится. Класс `Page` содержит только три базовых сеттера: `title=`, `body=` и `keywords=`.

Листинг 17.6. Использование класса `Page`. Файл page_use.rb

```
require_relative 'page'

page = Page.new

page.title = 'Страница'
page.body = 'Тело страницы'
page.keywords = ['Базовая страница']
page.phones = ['+7 920 4567722', '+7 920 4567733']

p page
```

Выполнение программы завершается ошибкой — неизвестный метод `phones=`:

```
page_use.rb:8:in `<main>': undefined method `phones='
```

17.2. Логические операторы

Классы предоставляют средства, позволяющие выяснить, является ли класс базовым или производным в отношении другого класса. Для этого можно использовать логические операторы: >, >=, <= и <. Для удобства загрузим классы в среду интерактивного Ruby (`irb`):

```
 > %w[about news photo_catalog page].each { |f| require_relative f }
=> ["about", "news", "photo_catalog", "page"]
 > About < Page
=> true
 > About > Page
=> false
 > About == About
=> true
 > About >= About
=> true
 > About >= Page
=> false
```

Класс `About` наследует от `Page`, поэтому сравнение `About < Page` возвращает «истину» `true`.

При попытке сравнения двух классов, которые не связаны отношением наследования, логические методы возвращают неопределенное значение `nil`:

```
 > About > News
=> nil
```

17.3. Динамический базовый класс

В роли базового класса может выступать любое Ruby-выражение, которое возвращает класс. В листинге 17.7 класс `Child` наследуется от Ruby-выражения, которое по случайному закону выдает либо класс `First`, либо класс `Second`.

Листинг 17.7. Случайный выбор базового класса. Файл base_rand.rb

```
class First; end
class Second; end

class Child < rand(0..1).zero? ? First : Second
end

puts Child.superclass
```

На практике, если в таком подходе возникает необходимость, Ruby-выражение, как правило, оформляют в виде метода (листинг 17.8).

Листинг 17.8. Случайный выбор базового класса. Файл base_method.rb

```
class First; end
class Second; end

COLLECTION =  { first: First, second: Second }

def get(klass)
  COLLECTION[klass] || Object
end

class Correct < get(:first); end
puts Correct.superclass # First

class Incorrect < get(:third); end
puts Incorrect.superclass # Object
```

В приведенном примере метод `get` возвращает класс `First` или `Second`, если в качестве аргумента ему передается символ `:first` или `:second`. Для любого другого аргумента метод возвращает класс `Object`.

17.4. Наследование констант

Константы, определенные в базовом классе, наследуются производными классами (листинг 17.9).

Листинг 17.9. Наследование констант. Файл constant.rb

```
class First
  CONST = 1
  COUNT = 10
end

class Second < First
  COUNT = 20
  def initialize
    puts CONST
  end

  def total
    COUNT
  end
end

second = Second.new # 1
puts second.total # 20

puts First::COUNT   # 10
puts Second::COUNT  # 20
```

Константа `CONST` определена здесь в классе `First`. К константе можно обращаться в производном классе `Second` так, как будто константа была определена внутри этого класса. При необходимости константы можно переопределять. Например, константа `COUNT` получает значение `10` на уровне класса `First` и значение `20` на уровне класса `Second`.

Как отмечалось в *главе 6*, при попытке переопределить константу Ruby-интерпретатор выводит предупреждение:

```
 > COUNT = 10
=> 10
 > COUNT = 20
(irb):2: warning: already initialized constant COUNT
(irb):1: warning: previous definition of COUNT was here
=> 20
```

Однако при переопределении константы в классе-наследнике предупреждение не выводится. В этом случае речь идет о двух разных константах: `First::COUNT` и `Second::COUNT`.

17.5. Иерархия стандартных классов

Узнать базовый класс можно при помощи метода `superclass`. Так, базовый класс для `About` — это `Page`. При помощи метода `superclass` можно восстановить всю цепочку наследования:

```
 > About.superclass
=> Page
 > About.superclass.superclass
=> Object
 > About.superclass.superclass.superclass
=> BasicObject
 > About.superclass.superclass.superclass.superclass
=> nil
```

Как можно видеть, обычный класс `Page` наследуется от класса `Object`, а тот, в свою очередь, от `BasicObject`. Класс `BasicObject` не имеет базового класса — это вершина иерархии (рис. 17.3).

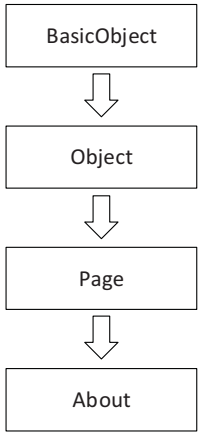


Рис. 17.3. Иерархия наследования классов

При создании классов мы каждый раз неявным образом наследуемся от класса `Object`. В листинге 17.10 закомментированный вариант определения класса `Page` полностью эквивалентен варианту с явным наследованием от `Object`.

Листинг 17.10. Явное наследование от класса `Object`. Файл page_object.rb

```
# class Page
#   attr_accessor :title, :body, :keywords
# end

class Page < Object
  attr_accessor :title, :body, :keywords
end
```

Так как явное наследование от `Object` необязательно, при создании классов всегда используется более короткий вариант.

До сих пор мы не сталкивались с классом `BasicObject`. При попытке создания объекта этого класса в консоли интерактивного Ruby (`irb`) будет получено сообщение о том, что метод `BasicObject` не поддерживает метод `inspect`, поэтому об этом объекте нечего рассказать:

```
 > o = BasicObject.new
(Object doesn't support #inspect)
=>
```

В объекте класса `BasicObject` отсутствуют почти все стандартные методы: `object_id`, `methods`, `inspect` и т. п.

Класс `Object`, от которого наследуются все остальные классы, весьма сильно перегружен методами. Именно поэтому в язык Ruby введен облегченный класс `BasicObject`, наследуясь от которого можно создавать облегченные иерархии.

Цепочку наследования классов можно получить более простым способом, если воспользоваться методом `ancestors`:

```
 > About.ancestors
=> [About, Page, Object, Kernel, BasicObject]
```

В качестве результата метод возвращает всю цепочку классов: от текущего класса `About` до `BasicObject`.

> ***Замечание***
>
> В цепочке наследования отображается модуль `Kernel`, который подмешивает на уровень класса `Object` глобальные методы. Более подробно модули и участие их в поиске методов рассматриваются в *главах 19–21*.

17.6. Переопределение методов

При наследовании можно не только использовать методы базового класса, но и переопределять их. В листинге 17.11 приводится модифицированный класс `Page`, в который добавлен метод `initialize`, позволяющий инициализировать инстанс-переменные класса `Page`.

Листинг 17.11. Добавление метода `initialize` в класс `Page`. Файл page_initialize.rb

```
class Page
  attr_accessor :title, :body, :keywords

  def initialize(title:, body:, keywords: [])
    @title = title
    @body = body
    @keywords = keywords
  end
end
```

В листинге 17.12 приводится пример использования обновленного класса `Page`.

Листинг 17.12. Использование класса `Page`. Файл page_initialize_use.rb

```
require_relative 'page_initialize'

page = Page.new title: 'Страница',
               body: 'Тело страницы',
               keywords: ['Базовая страница']

p page
```

Класс `News` содержит дополнительную инстанс-переменную `date`. Модифицируем метод `initialize` в производном классе таким образом, чтобы инстанс-переменная `date` автоматически получала текущую дату (листинг 17.13).

Листинг 17.13. Добавление метода `initialize` в класс `News`. Файл news_wrong.rb

```
require_relative 'page_initialize'

class News < Page
  attr_accessor :date

  def initialize(title:, body:, keywords: [])
    @date = Time.new
  end
end
```

Если попробовать воспользоваться классом `News` в таком виде, инстанс-переменные `@title`, `@body` и `@keywords` останутся неинициализированными.

Чтобы повторно не добавлять в метод `initialize` код инициализации из базового класса `Page`, можно воспользоваться ключевым словом `super`. Вызов этого ключевого слова аналогичен вызову одноименного метода базового класса, которому передаются все параметры текущего метода (листинг 17.14).

Листинг 17.14. Использование ключевого слова `super`. Файл super.rb

```
require_relative 'page_initialize'

class News < Page
  attr_accessor :date

  def initialize(title:, body:, keywords: [])
    @date = Time.new
    super
  end
end
```

В листинге 17.15 приводится пример создания объекта класса `News`.

Листинг 17.15. Использование класса `News`. Файл super_use.rb

```
require_relative 'super'

news = News.new title: 'Страница',
               body: 'Тело страницы',
               keywords: ['Базовая страница']

p news
```

Если запустить программу на выполнение, можно получить следующий отчет о состоянии объекта:

```
#<News:0x00007ff2540181c0 @date=2018-12-23 20:51:59 +0300, @title="Страница",
@body="Тело страницы", @keywords=["Базовая страница"]>
```

Если количество параметров в методе базового и производного классов не совпадает, можно явно задать аргументы ключевому слову `super`. В листинге 17.16 приводится вариант класса `News`, в котором метод `initialize` принимает в качестве параметра дату публикации.

Листинг 17.16. Явная передача параметров `super`. Файл super_params.rb

```
require_relative 'page_initialize'

class News < Page
  attr_accessor :date

  def initialize(title:, body:, date:, keywords: [])
    @date = date
    super(title: title, body: body, keywords: keywords)
  end
end
```

17.7. Удаление методов

При наследовании можно не только добавлять или заменять методы — Ruby позволяет удалять методы как в классах наследниках, так и во всей иерархии наследования. Для этого предназначены два метода: `undef_method` и `remove_method`. Оба они позволяют удалить метод из класса (листинг 17.17).

Листинг 17.17. Удаление методов. Файл delete_methods.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
```

```
    @date = date
  end

  def price
    @price
  end

  def date
    @date
  end
end

ticket = Ticket.new date: Time.new(2019, 5, 10, 10, 20)

p ticket.price # 500
p ticket.date  # 2019-05-10 10:20:00 +0300

class Ticket
  remove_method :price
  undef_method :date
end

p ticket.price # undefined method `price'
p ticket.date  # undefined method `date'
```

В приведенном примере создается класс `Ticket` с методом `initialize`, который позволяет инициализировать объект класса ценой (`@price`) и временем проведения мероприятия (`@date`). Два дополнительных метода: `price` и `date` — позволяют получить доступ к одноименным инстанс-переменным за пределами класса.

После создания объекта `ticket` класс `Ticket` повторно открывается, и из него удаляются методы `price` и `date`. Начиная с этого момента, все объекты класса лишаются удаленных методов и обратиться к ним не представляется возможным.

Удалять можно не только инстанс-методы, но и методы класса. Например, удалив из класса метод `new`, мы лишаем его возможности создавать объекты (листинг 17.18).

Листинг 17.18. Удаление метода класса `new`. Файл delete_new.rb

```
class Ticket
  remove_method :new
end

t = Ticket.new # method `new' not defined in Ticket (NameError)
```

Оба метода удаления выглядят похожими, однако между ними имеется различие. Метод `remove_method` удаляет метод лишь из текущего класса, в то время как метод `undef_method` вносит изменения во всю цепочку наследования (рис. 17.4).

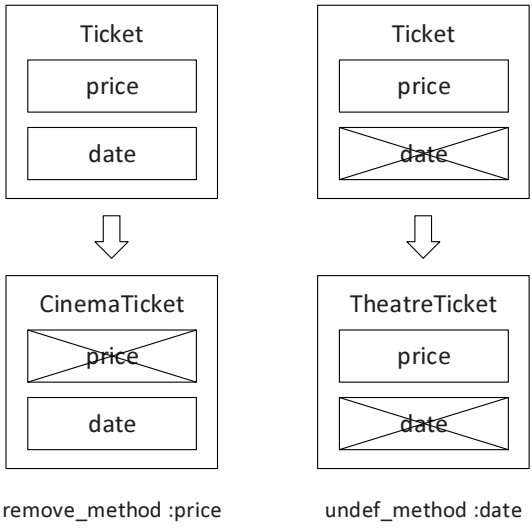


Рис. 17.4. Различие в способах удаления методов `remove_method` и `undef_method`

В листинге 17.19 приводится пример иерархии наследования — от базового класса билета `Ticket` наследуется два класса:

- `CinemaTicket` — билет в кино;
- `TheatreTicket` — билет в театр.

Класс `CinemaTicket` удаляет метод `price` при помощи `remove_method`. Класс `TheatreTicket` удаляет метод `date` при помощи `undef_method`.

Листинг 17.19. Различие методов `remove_method` и `undef_method`. Файл delete_diff.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
    @date = date
  end

  def price
    @price
  end

  def date
    @date
  end
end

ticket = Ticket.new date: Time.new(2019, 5, 10, 10, 20)

p ticket.price # 500
p ticket.date  # 2019-05-10 10:20:00 +0300
```

```
class CinemaTicket < Ticket
  def price
    super
  end
  remove_method :price
end

cinema = CinemaTicket.new price: 600, date: Time.new(2019, 5, 11, 10, 20)

p ticket.price # 500
p cinema.price # 600

class TheatreTicket < Ticket
  def price
    super
  end
  undef_method :date
end

theatre = TheatreTicket.new date: Time.new(2019, 5, 11, 10, 20)

p ticket.date  # 2019-05-10 10:20:00 +0300
p theatre.date # undefined method `date' for
```

Обратите внимание: несмотря на то, что метод `CinemaTicket#price` был удален, мы по-прежнему можем получать доступ к переменной `@pirce` через метод `Ticket#price`. Метод `remove_method` удаляет метод только в производном классе, не затрагивая методы базового класса.

При использовании метода `undef_method` удаляются все методы иерархии. Однако это относится только к объектам класса `TheatreTicket`, в котором был удален метод. Объекты базового класса `Ticket` остаются незатронутыми (рис. 17.5).

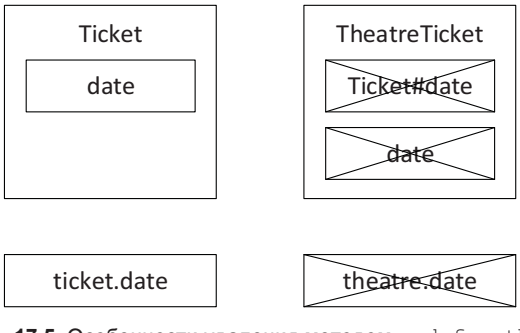


Рис. 17.5. Особенности удаления методом `undef_method`

17.8. Поиск метода

В *разд. 17.5* мы выяснили, что все классы выстраиваются в цепочку наследования. Искомый метод может оказаться в любом из классов из этой цепочки. Метод может быть определен в нескольких местах:

- метакласс объекта;
- класс объекта;
- базовый класс объекта;
- системный класс `Object`;
- системный класс `BasicObject`.

> ***Замечание***
>
> В *главе 20* будет показано, что в цепочку поиска метода дополнительно встраиваются модули.

При поиске метода Ruby-интерпретатор выбирает первый встретившийся ему метод. При этом поиск начинается с метакласса. Если в нем обнаруживается метод с подходящим названием — он будет вызван. Если метод обнаружить не удалось, поиск продолжается в классе объекта, потом в базовом классе. Поиск продолжается до тех пор, пока интерпретатор не доходит до вершины иерархии — класса `BasicObject` (рис. 17.6).

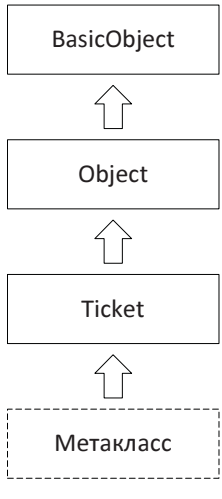


Рис. 17.6. Поиск метода

Из схемы, показанной на рис. 17.6, можно всегда понять, какой из методов в иерархии наследования будет вызван. В листинге 17.20 метод `say` определен как инстанс-метод класса `Ticket` и как синглетон-метод объекта `ticket`. Если мы сверимся со схемой на рис. 17.6, нетрудно догадаться, что Ruby-интерпретатор первым обнаружит синглетон-метод в метаклассе. До вызова `Ticket#say` дело не дойдет.

Листинг 17.20. Поиск метода say. Файл lookup_say.rb

```
class Ticket
  def say(name)
    "Ticket#say: Hello, #{name}!"
  end
end

ticket = Ticket.new

def ticket.say(name)
  "ticket.say: Hello, #{name}!"
end

puts ticket.say 'world' # ticket.say: Hello, world!
```

В листинге 17.21 переопределяется метод `object_id` для класса `About`. При поиске метода объект `about` использует метод `About#object_id`, а объекты `page` и `obj` — метод `Object#object_id`.

Листинг 17.21. Поиск метода object_id. Файл lookup_object_id.rb

```
class Page
  attr_accessor :title, :body, :keywords
end

class About < Page
  attr_accessor :phones, :address

  def object_id
    @object_id ||= rand(1..100)
  end
end

obj = Object.new
page = Page.new
about = About.new

puts obj.object_id    # 70273077078340
puts page.object_id   # 70273077078320
puts about.object_id  # 51
```

Теперь весьма легко понять, почему работает переопределение методов. Механизм поиска раньше обнаруживает переопределенный метод, чем метод в базовом или системном классах.

При удалении метода при помощи `remove_method` удаляется метод в текущем классе, открывая дорогу к методу в базовом. При этом `undef_method` удаляет методы и в текущем, и в базовых классах из всей иерархии классов.

Благодаря схеме поиска метода (см. рис. 17.6) можно понять, почему Ruby-разработчики всегда предпочитают `define_method` альтернативному `method_missing`. В случае `method_missing` может уходить весьма много времени на доказательство отсутствия метода в каждом из классов. Чем больше цепочка наследования, тем больше времени станет уходить на такое доказательство.

Задания

1. Создайте класс судна. Унаследуйте от него два типа судов: с возможностью плавать под водой и в надводном состоянии. С использованием полученных классов создайте:
   - атомную подводную лодку (ракеты, торпеды);
   - сухогруз для перевоза зерна (грузовой трюм, кран);
   - контейнеровоз (кран);
   - нефтяной танкер (грузовой трюм);
   - ракетный крейсер (ракеты);
   - военный транспорт (ракеты, грузовой трюм, кран).

   Подумайте, как лучше назвать классы, как организовать иерархию классов, чтобы сократить количество повторов в коде.

2. Создайте класс пользователя веб-сайта, позволяющий задать фамилию, имя, отчество и электронный адрес. Унаследуйте от него несколько классов:
   - обычный посетитель;
   - автор материала;
   - администратор;
   - модератор.

   Подумайте, как лучше назвать классы, как организовать иерархию классов и нужна ли она тут. Добавьте в каждый из классов инстанс-метод `say`, который должен сообщать роль пользователя. Переопределите метод `to_s`, который должен возвращать ту же строку, что и метод `say`. Постарайтесь максимально исключить повторы кода.

3. Создайте класс человека `Person`, от которого унаследуйте классы посетителей `User`, администраторов `Admin` и модераторов `Moderator`. Запретите возможность создания объектов класса `Person`. Всем объектам классов добавьте возможность задавать фамилию, имя и отчество пользователя, а также получать эти сведения.

4. Составьте иерархию классов наследования для Homo sapiens (Человек разумный):

   - Царство: Животные;
   - Тип: Хордовые;
   - Класс: Млекопитающие;
   - Отряд: Приматы;
   - Семейство: Гоминиды;
   - Род: Люди;
   - Вид: Человек разумный.

   Создайте объекты этих классов.

ГЛАВА 18

Области видимости

Файлы с исходными кодами этой главы находятся в каталоге *scope* сопровождающего книгу электронного архива.

Методы создают границу для переменных: локальные переменные не могут пересекать эту границу, в то время как для инстанс-переменных, глобальных и классов переменных эта граница прозрачна.

Классы выступают границей для методов. По умолчанию методы открыты, но область их видимости можно ограничить, объявив закрытыми или защищенными. Эта глава посвящена деталям ограничения области видимости методов.

18.1. Концепция видимости

В объектно-ориентированных языках программирования методы принято делить на:

- `public` — открытые;
- `private` — закрытые;
- `protected` — защищенные.

Практически все объектно-ориентированные языки программирования реализуют эту концепцию.

До сих пор мы часто использовали глобальный метод `puts`:

```
 > puts 'Hello, world!'
Hello, world!
=> nil
```

В то же время не раз подчеркивалось, что методы не могут быть вызваны без получателя. Если получатель не указан явно, предполагается, что в качестве получателя выступает ссылка на текущий объект `self`. Однако при попытке указать получатель явно, нас ждет неудача, вызов завершается ошибкой:

```
> self.puts 'Hello, world!'
NoMethodError (private method `puts' called for main:Object)
```

Метод оказался закрытым, и вызывать его таким способом не получится! Поэтому важно понимать концепцию открытых, закрытых и защищенных методов.

18.2. Открытые методы

По умолчанию все создаваемые методы являются открытыми. В листинге 18.1 представлен класс `Ticket`, в котором реализованы методы: `initialize`, `price` и `date`. Метод `price` возвращает цену билета, а `date` — время и дату.

Листинг 18.1. Класс с открытыми по умолчанию методами. Файл ticket_public.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
    @date = date
  end

  def price
    @price
  end

  def date
    @date
  end
end
```

После создания объекта класса `Ticket` можно обращаться к его методам `price` и `date`, т. к. по умолчанию все методы класса являются *открытыми* (листинг 18.2).

Листинг 18.2. Обращение к открытым методам. Файл ticket_public_use.rb

```
require_relative 'ticket_public'

ticket = Ticket.new date: Time.mktime(2019, 5, 10, 10, 20)

puts ticket.price # 500
puts ticket.date  # 2019-05-10 10:20:00 +0300
```

Методы `price` и `date` можно было бы объявить открытыми при помощи метода `public`. Ему передаются названия методов, которые необходимо объявить открытыми (листинг 18.3).

Листинг 18.3. Явное использование метода `public`. Файл method_public.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
```

```
    @date = date
  end

  def price
    @price
  end

  def date
    @date
  end

  public :price, :date
end
```

У этого метода есть и другая форма записи — можно вызывать метод `public` без аргументов (листинг 18.4), и тогда все объявленные после его вызова методы считаются открытыми — до тех пор, пока не встретится другой метод, управляющий областью видимости: `private` или `protected`. Поскольку методы по умолчанию и так открыты, метод `public` крайне редко встречается на практике.

Листинг 18.4. Альтернативный способ вызова `public`. Файл method_public_alter.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
    @date = date
  end

  public

  def price
    @price
  end

  def date
    @date
  end
end
```

18.3. Закрытые методы

Методы можно объявить закрытыми, для чего следует воспользоваться методом `private`. Такие методы нельзя вызывать за пределами класса.

В листинге 18.5 метод `price` будет доступен для вызова, т. к. все методы класса по умолчанию открыты. Метод `date` окажется закрытым, т. к. он расположен после вызова `private`.

Листинг 18.5. Использование метода `private`. Файл ticket_private.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
    @date = date
  end

  def price
    @price
  end

  private

  def date
    @date
  end
end
```

При попытке получить доступ к закрытому методу `date` будет выведено сообщение об ошибке (листинг 18.6).

Листинг 18.6. Обращение к закрытому методу. Файл ticket_private_use.rb

```
require_relative 'ticket_private'

ticket = Ticket.new date: Time.mktime(2019, 5, 10, 10, 20)

puts ticket.price # 500
puts ticket.date  # private method `date' called for
#<Ticket:0x00007f95fb996298>
```

Как правило, закрытыми объявляют методы, которые используются лишь другими методами класса. Например, можно объявить вспомогательный метод `price_format`, который будет задействован в методе `price` для форматирования цены.

Поскольку метод `price_format` предназначен лишь для внутреннего пользования, мы объявляем его закрытым (листинг 18.7). Тем самым внешним разработчикам посылается сигнал, что метод не является частью открытого интерфейса класса и может быть изменен в следующих версиях класса.

Листинг 18.7. Закрытый метод `price_format`. Файл price_format.rb

```
class Ticket
  def initialize(date:, price: 500)
    @price = price
    @date = date
  end
```

```
  def price
    price_format(@price)
  end

  def date
    @date
  end

  private

  def price_format(price)
    format('Цена билета %.f', price)
  end
end
```

У закрытых методов нельзя указать получатель, даже `self`. Следующий вызов всегда будет приводить к ошибке:

```
def price
  self.price_format(@price)
end
```

Именно поэтому заканчивается ошибкой попытка вызова метода `puts` с явным префиксом `self.`:

```
> self.puts 'Hello, world!'
NoMethodError (private method `puts' called for main:Object)
```

18.4. Защищенные методы

Защищенные методы занимают промежуточное положение между закрытыми и открытыми. Как и закрытые методы, их нельзя использовать за пределами класса. Однако внутри класса допускается применять их совместно с получателем. В качестве получателя может выступать ключевое слово `self` или объект класса, в котором определен закрытый метод.

В листинге 18.8 представлен класс `Person`, представляющий игрока в игре. Класс содержит две инстанс-переменные: имя игрока — `@name` и его счет в игре — `@score`.

Листинг 18.8. Класс игрока `Person`. Файл person.rb

```
class Person
  attr_accessor :name

  def initialize(name:, score:)
    @name = name
    @score = score
  end
```

```
  private

  def score
    @score
  end
end
```

В листинге 18.9 создаются два игрока: `first` и `second`, на счетах у которых 12 и 8 баллов соответственно. В программе сравниваются счета игроков, чтобы определить победителя. Попытка обратиться к методу `score` ожидаемо заканчивается неудачей, т. к. метод закрыт.

Листинг 18.9. Использование класса `Person`. Файл person_use.rb

```
require_relative 'person'

first  = Person.new(name: 'Первый игрок', score: 12)
second = Person.new(name: 'Второй игрок', score: 8)

if first.score > second.score
  puts 'Выигрывает первый игрок'
elsif first.score < second.score
  puts 'Выигрывает второй игрок'
else
  puts 'Ничья'
end
```

Замена метода `private` на `protected` в классе `Person` не позволит воспользоваться методом `score`, поскольку защищенные методы нельзя использовать за пределами класса, и сравнение придется перенести внутрь класса.

Помочь может перегрузка оператора сравнения `<=>`. Оператор возвращает: `1` — если первый операнд больше второго, `-1` — если первый операнд меньше второго, и `0` — если операнды равны (листинг 18.10).

Листинг 18.10. Перегрузка оператора <=>. Файл person_compare.rb

```
class Person
  attr_accessor :name

  def initialize(name:, score:)
    @name = name
    @score = score
  end

  def <=>(person)
    score <=> person.score
  end
```

```
  protected

  def score
    @score
  end
end
```

Метод <=> принимает в качестве параметра другой объект класса Person. Внутри метода производится сравнение игрового результата score текущего объекта и защищенного метода person.score другого объекта.

В листинге 18.11 приводится пример использования обновленного класса Person.

Листинг 18.11. Использование класса Person. Файл person_compare_use.rb

```
require_relative 'person_compare'

first  = Person.new(name: 'Первый игрок', score: 12)
second = Person.new(name: 'Второй игрок', score: 8)

case first <=> second
when 1 then puts 'Выигрывает первый игрок'
when -1 then puts 'Выигрывает второй игрок'
when 0 then puts 'Ничья'
end
```

Результатом выполнения программы будет сообщение о том, что выигрывает первый игрок.

18.5. Закрытый конструктор

Иногда необходимо закрыть или защитить методы класса. Один из наиболее часто используемых методов класса — это метод new, при помощи которого создаются объекты класса. Основная задача этого метода — инициализация объекта (см. *разд. 14.2.5*).

Доступ к методу new ограничивают, если необходимо наложить ограничения на процесс создания объектов (например, разрешается создать только пять объектов). В таком случае обычно запрещается прямое создание объекта путем закрытия метода new. Для создания объектов используется другой метод, в котором реализована логика ограничения. В листинге 18.12 объявляется класс Person, открывается метакласс объекта Person и при помощи метода private закрывается метод new.

Листинг 18.12. Ограничение доступа к методу new. Файл person_private_new.rb

```
class Person
  class << self
```

```
    private :new
  end
end

user = Person.new # private method `new' called for Person:Class
```

Конструкция в листинге 18.12 получилась неуклюжая. Вместо нее Ruby предоставляет метод `private_class_method`, при помощи которого можно объявлять методы класса закрытыми (листинг 18.13).

> ***Замечание***
>
> Помимо метода `private_class_method` Ruby предоставляет методы `public_class_method` и `protected_class_method`, которые позволяют объявить метод класса открытым или защищенным.

Листинг 18.13. Использование `private_class_method`. Файл private_class_method.rb

```
class Person
  private_class_method :new
end

user = Person.new # private method `new' called for Person:Class
```

18.6. Паттерн «Одиночка» (*Singleton*)

Вариантов использования объектно-ориентированного программирования очень много. Не все подходы приводят к созданию надежных и легко сопровождаемых проектов. За десятилетия разработки были выявлены наиболее удачные приемы, которые называются *паттернами*, или *шаблонами проектирования*.

Каждый из паттернов имеет звучное, хорошо запоминающееся название, которое разработчики используют для быстрого и емкого обозначения типового решения.

Паттерн «Одиночка», или «Синглетон» (`Singleton`) — это пример шаблона проектирования, где оказывается востребованным закрытый конструктор. Паттерн проектирования не следует путать с синглетон-методом. В данном случае имеется в виду класс, объект которого может существовать в одном экземпляре. Это может быть соединение с базой данных или объект с настройками сайта. Такие объекты не должны существовать в двух или более экземплярах, чтобы предотвратить ситуацию, когда редактируется один экземпляр, а используется другой.

В листинге 18.14 приводится одна из возможных реализаций паттерна «Синглетон».

> ***Замечание***
>
> В *главе 21* будет показан более простой способ реализации паттерна «Синглетон» при помощи стандартного модуля `Singleton`.

Листинг 18.14. Реализация паттерна проектирования «Синглетон». Файл singleton.rb

```
class Singleton
  def self.instance
    @@obj ||= new
  end

  def dup
    @@obj
  end

  private_class_method :new
  alias clone dup
end

first = Singleton.instance
p first        # #<Singleton:0x00007fadc29a1b00>
second = Singleton.instance
p second       # #<Singleton:0x00007fadc29a1b00>

p first.dup    # #<Singleton:0x00007fadc29a1b00>
p first.clone  # #<Singleton:0x00007fadc29a1b00>

third = Singleton.new # private method `new' called for Singleton:Class
```

Для того чтобы предотвратить создание нескольких объектов класса `Singleton`, метод `new` объявляется закрытым при помощи `private_class_method`.

Поэтому создать объект класса привычными средствами не получится. Для получения объекта предусмотрен метод класса `instance`. Метод создает объект при помощи закрытого метода `new`, если переменная класса `@@obj` имеет неопределенное значение. Если метод `instance` уже вызывался, переменная `@@obj` ссылается на объект. При помощи оператора ||= предотвращается вызов метода `new`, если объект уже получен.

Поэтому при каждом вызове `instance` будет возвращаться ссылка на один и тот же объект. Чтобы предотвратить клонирование объекта, метод `dup` переопределяется, а метод `clone` объявляется псевдонимом `dup`.

Теперь, какие бы манипуляции мы ни совершали с объектом `Singleton` и объектом, — нам не удастся создать более одного объекта.

Сейчас класс `Singleton` не несет никакой полезной нагрузки. Для того чтобы его объект можно было использовать, — например, для хранения настроек, — придется расширить его функционал.

Переименуем класс в `Settings` и добавим инстанс-переменную `@list`, которая будет хранить хэш с настройками, сохраняемыми пользователем. Для того чтобы в переменной можно было сохранить множество значений, сделаем переменную `@list` хэшем. Удобнее всего для этого воспользоваться методом `initialize`.

Кроме того, перегрузим для объекта класса `Settings` квадратные скобки, адресовав инстанс-переменной `@list` методы `[]` и `[]=`. Это позволит обращаться с объектом как с хэшем (листинг 18.15).

Листинг 18.15. Класс `Settings`. Файл settings.rb

```
class Settings
  def initialize
    @list = {}
  end

  def self.instance
    @@obj ||= new
  end

  def dup
    @@obj
  end

  def [](key)
    @list[key]
  end

  def []=(key, value)
    @list[key] = value
  end

  private_class_method :new
  alias clone dup
end
```

Теперь можно воспользоваться классом `Settings` для хранения настроек. И не опасаться при этом появления копии объекта — он всегда будет существовать в единственном экземпляре (листинг 18.16).

Листинг 18.16. Использование класса `Settings`. Файл settings_use.rb

```
require_relative 'settings'

setting = Settings.instance
setting[:title] = 'Новостной портал'
setting[:per_page] = 30

params = Settings.instance
p params[:title]    # "Новостной портал"
p params[:per_page] # 30
```

18.7. Вызов закрытых методов

Иногда требуется обойти запрет на вызов закрытых методов. Например, удалить уже созданную константу нельзя. При попытке назначить константе новое значение Ruby-интерпретатор выводит предупреждение (листинг 18.17).

Листинг 18.17. Переопределение константы. Файл constants_redefine.rb

```
CONST = 1
CONST = 2
puts CONST # 2
```

Запуск программы сопровождается предупреждением, что константа уже определена:

```
constants_redefine.rb:2: warning: already initialized constant CONST
constants_redefine.rb:1: warning: previous definition of CONST was here
2
```

Выходом из ситуации стала бы возможность удаления существующей константы перед ее повторным определением. Для этого можно было бы воспользоваться методом `Object::remove_const`, однако метод является закрытым. Попытка вызова его завершится неудачей:

```
CONST = 1
Object.remove_const :CONST
CONST = 2
puts CONST
```

Существует обходной путь вызова закрытых методов — для этого можно воспользоваться методом `send` (листинг 18.18).

Листинг 18.18. Вызов закрытого метода. Файл constants_send.rb

```
CONST = 1
Object.send(:remove_const, :CONST)
CONST = 2
puts CONST # 2
```

Метод `send` не проверяет области видимости, поэтому с его помощью можно вызывать любой метод, лишь бы он существовал.

Разумеется, вызов закрытых методов — довольно-таки «грязный» прием. Методы объявляют закрытыми не для того, чтобы ими пользовались за пределами класса.

С другой стороны, `send` демонстрирует, что защита и закрытие методов в Ruby весьма условны.

> ***Замечание***
>
> Реализовать закрытые методы в полностью объектно-ориентированных языках программирования — таких как Ruby или Python — довольно трудно. Например, в Python вообще нет аналогов методов `public`, `private` и `protected`.

Если важно сохранить области видимости, применяя метод `send`, можно воспользоваться аналогичным методом `public_send`. Он действует точно так же, как `send`, однако не позволяет вызывать закрытые и защищенные методы (листинг 18.19).

Листинг 18.19. Использование метода `public_send`. Файл public_send.rb

```
puts Object.public_send(:object_id)

CONST = 1
Object.public_send(:remove_const, :CONST)
```

Результатами работы этой программы будут успешный вызов метода `Object.object_id` и ошибка при попытке обратиться к закрытому методу `remove_const`:

```
70218827615200
`public_send': private method `remove_const' called for Object:Class
```

18.8. Информационные методы

Как уже отмечалось, методы в классе могут быть открытыми, защищенными и закрытыми. Открытые методы доступны как внутри класса, так и за его пределами. Закрытые методы доступны только внутри класса и не могут быть вызваны с получателем, даже `self`. Защищенные методы могут быть вызваны только внутри класса, но для них допускается использовать ограниченный набор получателей: `self` и объект того же класса (рис. 18.1).

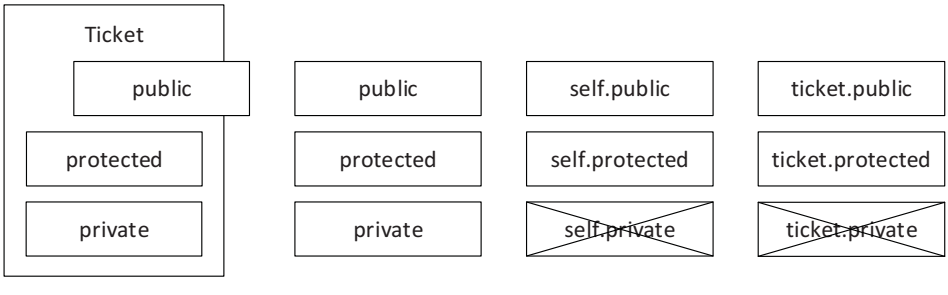


Рис. 18.1. Открытые, закрытые и защищенные методы

Получить список методов класса можно при помощи уже знакомого нам метода `methods`. В том случае, если необходимо получить три отдельных списка с открытыми, защищенными и закрытыми методами, можно воспользоваться методами `public_methods`, `protected_methods` и `private_methods`. Для того чтобы продемонстрировать работу этих методов, создадим класс `Ticket`, который будет содержать открытые, защищенные и открытые методы (листинг 18.20).

Листинг 18.20. Класс `Ticket`. Файл ticket.rb

```
class Ticket
  attr_accessor :price, :date
  @@next_number = 1

  def initialize(date:, price: 500)
    @date = date
    @price = price
    @number = @@next_number
    @@next_number += 1
  end

  def <=>(ticket)
    number <=> ticket.number
  end

  def price
    price_format(@price)
  end

  private

  def price_format(price)
    format('%.f', price)
  end

  protected

  def number
    @number
  end
end
```

Класс содержит три инстанс-переменные: цену билета — `@price`, дату проведения мероприятия — `@date` и уникальный номер билета — `@number`. Для цены и даты при помощи `attr_accessor` создаются геттеры и сеттеры. Метод-геттер для цены переопределен таким образом, чтобы цена возвращалась с точностью до второго знака после запятой. Для формирования цены используется закрытый метод `price_format`.

Обеспечить каждому билету его собственный уникальный номер помогает переменная класса `@@next_number`, которая увеличивается на единицу при каждом вызове метода `initialize`. Это значение присваивается инстанс-переменной `@number`.

Получить доступ к переменной `@number` за пределами класса нельзя — метод-геттер `number` объявлен защищенным и используется лишь в переопределенном операторе `<=>`. При сравнении билет считается «бóльшим», если у него больше значение `@number`.

В листинге 18.21 приводится пример использования объектов класса `Ticket`.

Листинг 18.21. Использование класса `Ticket`. Файл ticket_use.rb

```
require_relative 'ticket'

first = Ticket.new(date: Time.mktime(2019, 5, 10, 10, 20))
second = Ticket.new(date: Time.mktime(2019, 5, 10, 10, 20))

puts first <=> second # -1
```

Класс `Ticket` содержит открытые, защищенные и закрытые методы. Поэтому на его объектах хорошо видна разница между вызовами методов `public_methods`, `protected_methods` и `private_methods`. Чтобы исключить длинный список всех предопределенных методов, в листинге 18.22 методам передается аргумент `false`, который требует, чтобы результат содержал только определенные в классе `Ticket` методы.

Листинг 18.22. Файл ticket_methods.rb

```
require_relative 'ticket'

ticket = Ticket.new date: Time.mktime(2019, 5, 10, 10, 20)

p ticket.public_methods(false)    # [:price, :<=>, :price=, :date=, :date]
p ticket.protected_methods(false) # [:number]
p ticket.private_methods(false)   # [:price_format, :initialize]
```

Метод `initialize` является закрытым, несмотря на то, что мы не закрывали его явно.

Ruby предоставляет аналоги методов класса для методов `public_methods`, `protected_methods` и `private_methods`. Можно не создавать объект класса `Ticket`, а вызывать в отношении этого класса методы `public_instance_methods`, `protected_instance_methods` и `private_instance_methods` (листинг 18.23).

Листинг 18.23. Файл ticket_instance_methods.rb

```
require_relative 'ticket'

p Ticket.public_instance_methods(false)
# [:price, :<=>, :price=, :date=, :date]
p Ticket.protected_instance_methods(false)
# [:number]
p Ticket.private_instance_methods(false)
# [:price_format, :initialize]
```

18.9. Области видимости при наследовании

При наследовании классов области их видимости остаются прежними, если методы не переопределяются в производном классе с новой областью видимости. В листинге 18.24 приводится пример класса `TicketChild`, который наследует методы от определенного ранее класса `Ticket` (листинг 18.20).

Листинг 18.24. Области видимости при наследовании. Файл ticket_child.rb

```
require_relative 'ticket'

class TicketChild < Ticket
  def price
    price_format(@price)
  end

  def number
    super
  end
end

first = TicketChild.new(date: Time.mktime(2019, 5, 10, 10, 20))
second = TicketChild.new(date: Time.mktime(2019, 5, 10, 10, 20))

puts first <=> second         # -1

puts second.price             # 500
puts second.number            # 2
puts second.price_format(600) # private method `price_format' called
```

Внутри класса по-прежнему можно использовать закрытые и защищенные методы. За пределами класса попытка обратиться к закрытому методу `price_format` заканчивается ошибкой. Однако закрытый на уровне класса `Ticket` метод `number` был объявлен открытым в классе `TicketChild`. Поэтому у объектов производного класса `TicketChild` можно узнать уникальный номер билета, обратившись к методу `number`.

Для того чтобы определить, является ли метод открытым, защищенным или закрытым, можно использовать логические методы `public_method_defined?`, `protected_method_defined?` и `private_method_defined?`:

```
 > require_relative 'ticket_child.rb'
...
 > Ticket.public_method_defined? :number
=> false
 > Ticket.protected_method_defined? :number
=> true
```

```
 > Ticket.private_method_defined? :number
=> false
 > TicketChild.public_method_defined? :number
=> true
 > TicketChild.protected_method_defined? :number
=> false
 > TicketChild.private_method_defined? :number
=> false
```

18.10. Области видимости методов класса

До сих пор мы рассматривали область видимости лишь инстанс-методов. Однако `public`, `protected` и `private` можно применять и к методам класса.

В листинге 18.25 приводится пример класса `Ticket`, в котором количество созданных объектов регистрируется в переменной класса `@@count`. Значение переменной увеличивается на единицу при каждом вызове метода `initialize`.

Листинг 18.25. Файл singleton_methods.rb

```
class Ticket
  @@count = 0

  def initialize
    @@count += 1
  end

  def count
    Ticket.count
  end

  class << self

    def report
      "Продано билетов: #{count}"
    end

    private

    def count
      @@count
    end
  end
end

first = Ticket.new
second = Ticket.new
```

```
puts Ticket.report # Продано билетов: 2
puts Ticket.count # private method `count' called for Ticket:Class
puts first.count # private method `count' called for Ticket:Class
```

Для доступа к переменной `@@count` создается метод класса `count`, который объявляется закрытым при помощи `private`.

В результате открытый метод класса `report` имеет доступ `count`, а попытки обратиться к нему за пределами класса заканчиваются ошибкой.

Более того, к этому методу нельзя обратиться даже из инстанс-методов класса. Класс `Ticket` и его метакласс — это два разных класса с разным набором методов, разным размещением в оперативной памяти и с разными ссылками `self` (рис. 18.2).

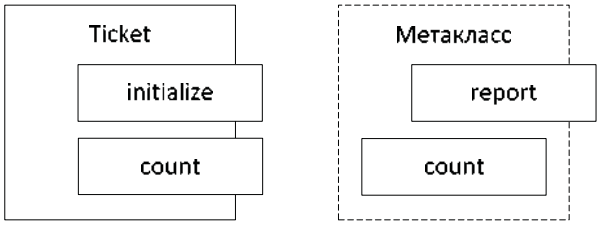


Рис. 18.2. Класс `Ticket` и его метакласс

Для того чтобы закрыть методы класса, `private`, `protected` и `private` нужно вызывать именно на уровне метакласса. Если заменить конструкцию `class << self` на определение методов через `self.count` и `self.report`, методы класса превратятся в открытые. Вызов `private` на уровне класса `Ticket` не будет действовать на уровне метакласса (листинг 18.26).

Листинг 18.26. Файл singleton_methods_wrong.rb

```
class Ticket
  @@count = 0

  def initialize
    @@count += 1
  end

  def count
    Ticket.count
  end

  def self.report
    "Продано билетов: #{count}"
  end

  private # не подействует на self.count

  def self.count
    @@count
```

```
  end
end

first = Ticket.new
second = Ticket.new

puts Ticket.report # Продано билетов: 2
puts Ticket.count # 2
puts first.count # 2
```

В результате все методы класса `Ticket` оказались открытыми. Прямая замена `private` на `private_class_method` тоже не поможет, т. к. мы находимся в контексте класса `Ticket`. Потребуется явная передача названия метода в качестве аргумента `private_class_method` (листинг 18.27).

Листинг 18.27. Файл singleton_methods_private.rb

```
class Ticket
  @@count = 0

  def initialize
    @@count += 1
  end

  def count
    Ticket.count
  end

  def self.report
    "Продано билетов: #{count}"
  end

  def self.count
    @@count
  end

  private_class_method :count
end
...
```

Задания

1. Изучите паттерн проектирования «Стратегия», реализуйте его на Ruby.
2. Изучите паттерн проектирования «Наблюдатель», реализуйте его на Ruby.
3. Изучите паттерн проектирования «Декоратор», реализуйте его на Ruby.

ГЛАВА 19

Модули

Файлы с исходными кодами этой главы находятся в каталоге *namespaces* сопровождающего книгу электронного архива.

Модуль — это синтаксическая конструкция, которая выполняет несколько функций. В первую очередь, модули обеспечивают пространство имен. Они также могут выступать в роли контейнеров методов. Кроме того, модули играют важную роль в объектно-ориентированном программировании и поиске методов.

Модулям посвящено несколько следующих глав. В этой главе мы рассмотрим модули с точки зрения механизма, обеспечивающего пространство имен в Ruby.

19.1. Создание модуля

Для создания модуля предназначено ключевое слово `module`, после которого указывается имя в `CamelCase`-стиле. Далее следует тело модуля, которое продолжается до тех пор, пока не встретится завершающее ключевое слово `end` (листинг 19.1).

Листинг 19.1. Создание модуля. Файл module.rb

```
module Site
end
```

В теле модуля можно размещать константы, методы, классы и другие модули (листинг 19.2). Как и в классах, допускается использование в теле модуля любых Ruby-выражений.

Листинг 19.2. Размещение объявлений в теле модуля. Файл module_body.rb

```
module Site
  VERSION = '1.1.0'

  def greeting(str)
    "Hello, #{str}!"
  end
```

```
  class Settings
  end
end
```

Созданный нами модуль `Site` можно заменить классом — для этого достаточно использовать вместо ключевого слова `module` ключевое слово `class` (листинг 19.3).

Листинг 19.3. Класс `Site`. Файл class_site.rb

```
class Site
  VERSION = '1.1.0'

  def greeting(str)
    "Hello, #{str}!"
  end

  class Settings
  end
end
```

Как будет показано в этой главе далее, классы и модули часто взаимозаменяемы и обладают схожими свойствами. Однако модули, в отличие от классов, не допускают создания объектов — у них отсутствует метод `new`:

```
 > require_relative 'module'
=> true
 > Site.new
NoMethodError (undefined method `new' for Site:Module)
```

Поэтому в Ruby-сообществе принято соглашение: классы используются в тех случаях, когда требуется создание объекта класса при помощи метода `new`. Если предполагается создание класса для организации пространства имен, для изоляции констант, методов и других классов, вместо класса лучше создать модуль.

19.2. Оператор разрешения области видимости

Для того чтобы обратиться к константе или классу, обычно используют оператор разрешения области видимости `::`. Слева от оператора указывается имя модуля, справа — объект в теле модуля.

В листинге 19.2 в теле модуля `Site` определены константа `VERSION` и класс `Settings`. Обратиться к ним можно при помощи оператора `::` (листинг 19.4).

Листинг 19.4. Оператор разрешения области видимости. Файл module_body_use.rb

```
require_relative 'module_body'

puts Site::VERSION        # 1.1.0
```

```
obj = Site::Settings.new
p obj.object_id          # 70235562723600
```

Так как имя модуля задается с прописной (заглавной) буквы, его можно легко импортировать при помощи методов `require` и `require_relative`. Поэтому `require_relative` в первой строке программы легко импортирует модуль `Site` (см. листинг 19.2).

Помимо константы `VERSION` и класса `Settings`, в модуле `Site` определен метод `greeting`. Однако обратиться к нему напрямую не получится ни при помощи оператора «точка», ни при помощи оператора разрешения области видимости.

Чтобы обратиться к методу, потребуется либо подмешать модуль в состав класса (см. *главу 20*), либо преобразовать метод в синглетон-метод модуля (см. *разд. 14.4*).

19.3. Пространство имен

Пространство имен — это именованный фрагмент программы, содержащий константы, методы, классы и модули. Пространство имен организовывает код проекта в иерархию, напоминающую файловую систему. Как файлы с одинаковыми именами изолированы, если находятся в разных каталогах, так и классы, методы и константы Ruby изолированы в разных пространствах имен. Это позволяет избегать конфликтов со сторонними библиотеками, а также облегчает поиск и загрузку классов.

В Ruby пространство имен организуется при помощи вложенных модулей и классов (листинг 19.5).

Листинг 19.5. Изоляция кода в пространстве имен. Файл my_namespace.rb

```
module MyNamespace
  class Array
    def to_s
      'my class'
    end
  end
end

p Array.new                 # []
p MyNamespace::Array.new    # #<MyNamespace::Array:0x00007fe7ea9d1c08>

puts Array.new              # nil
puts MyNamespace::Array.new # my class
```

В приведенном примере пользовательский класс `Array` помещен в модуль `MyNamespace`. Если объявить класс за пределами модуля `MyNamespece`, окажется открыт и изменен стандартный класс `Array`. В результате поведение всех массивов

будет изменено. Использование пространства имен позволяет исключить случайное изменение уже существующего класса.

19.4. Вложенные классы и модули

Вложенные классы и модули не обязательно используются для изоляции кода. Иногда вложенные модули могут быть задействованы для группировки подсистем большой программы.

Создадим класс `BattleField`, который моделирует квадратное игровое поле, состоящее из клеток. По умолчанию игровое поле будет использоваться для игры в шахматы, однако при желании оно может быть адаптировано для любой другой игры, в которой требуется квадратное игровое поле. Клеток может быть 10×10 — для игры в «Морской бой» или 8×8 — для игры в шахматы (рис. 19.1).

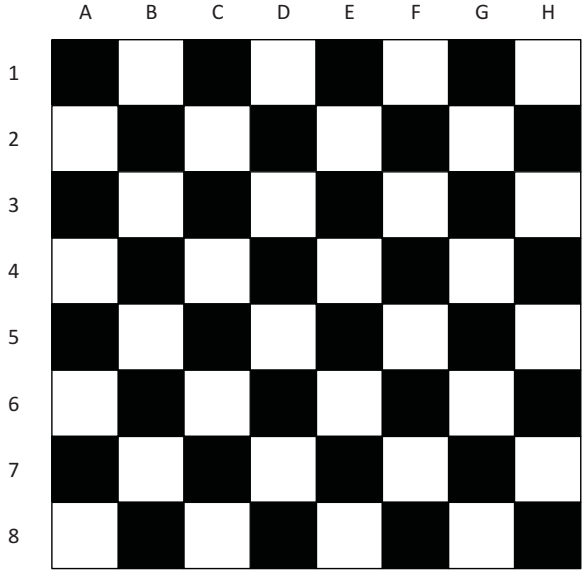


Рис. 19.1. Шахматная доска 8×8

Будущий класс `BattleFiled` можно разбить на три части:

- основной класс игрового поля `BattleFiled`;
- вспомогательный класс клетки поля `Filed`;
- модуль `Chess` с константами и методами, которые относятся к шахматам.

Возможная реализация класса `BattleFiled` представлена в листинге 19.6. Так как классы `Field` и `Chess` не используются за пределами `BattleFiled`, они реализованы в виде вложенных класса и модуля.

Метод `BattleField#initialize` принимает единственный параметр `size`, который задает размер поля. По умолчанию размер поля задается константой `BattleField::`

`Chess::SIZE`. В модуле `Chess` сосредоточены константы, которые относятся к шахматам: `X` — буквенные обозначения одной из сторон игрового поля, `GAMERS` — цвета фигур: белые — `white` и черные — `black`.

Листинг 19.6. Класс игрового поля `BattleField`. Файл battle_field.rb

```
class BattleField
  attr_accessor :size, :fields

  def initialize(size: BattleField::Chess::SIZE)
    @size = size
    @fields = yield(size) if block_given?
    @fields ||= Array.new(size) do |y|
                  Array.new(size) do |x|
                    Field.new(x: Chess::X[x], y: y + 1)
                  end
                end
  end

  def to_s
    lines.join("\n")
  end

  private

  def lines
    @fields.map do |line|
      line.map(&:to_s).join ' '
    end
  end

  class Field
    attr_accessor :x, :y

    def initialize(x:, y:)
      @x = x
      @y = y
    end

    def to_s
      "#{x}:#{y}"
    end
  end

  module Chess
    SIZE = 8
```

```
    X = %w[A B C D E F G H]
    GAMERS = [:white, :black]
  end
end
```

Каждая клетка игрового поля задана объектом класса `Field`, который содержит две инстанс-переменные: `@x` и `@y` — для хранения координат поля на игровой доске. Кроме того, класс задает метод `to_s` — для строкового представления точки в виде строки вида `'E:4'`.

В методе `BattleField#initialize` по умолчанию инстанс-переменной `@fields` присваивается массив 8×8 объектов класса `Filed`. Для инициализации массива мы вынуждены использовать блоки, чтобы каждое поле получило индивидуальный объект (см. *разд. 14.2.2*).

При помощи метода `block_given?` проверяется, передан ли методу `BattleField#initialize` блок. В случае, если метод возвращает «истину», вместо создания массива вызывается ключевое слово `yield`, через который в блок передается размер поля `size`. Переменная `@fields` инициализируется значением, которое возвращает блок.

Класс `BattleField` реализует метод `to_s` для вывода строкового представления игрового поля. Метод `to_s` использует вспомогательный метод `lines`, который превращает двумерный массив полей в одномерный массив строк. Таким образом, в `to_s` остается только соединить строки при помощи перевода строки `\n` (см. *разд. 2.4.2*).

В листинге 19.7 приводится пример использования класса `BattleField`, формируется шахматное поле и выводится его строковое представление.

Листинг 19.7. Шахматная доска. Файл battle_field_chess.rb

```
require_relative 'battle_field'

fields = BattleField.new
puts fields
```

Результатом выполнения программы будет следующее представление шахматной доски:

```
A:1 B:1 C:1 D:1 E:1 F:1 G:1 H:1
A:2 B:2 C:2 D:2 E:2 F:2 G:2 H:2
A:3 B:3 C:3 D:3 E:3 F:3 G:3 H:3
A:4 B:4 C:4 D:4 E:4 F:4 G:4 H:4
A:5 B:5 C:5 D:5 E:5 F:5 G:5 H:5
A:6 B:6 C:6 D:6 E:6 F:6 G:6 H:6
A:7 B:7 C:7 D:7 E:7 F:7 G:7 H:7
A:8 B:8 C:8 D:8 E:8 F:8 G:8 H:8
```

По аналогии с модулем `BattleField::Chess` можно создавать модули, относящиеся к другим играм. В листинге 19.8 приводится пример модуля `BattleFiled::Ship` для игры в «Морской бой».

Листинг 19.7. Модуль для «Морского боя». Файл battle_filed_ship_open.rb

```
require_relative 'battle_field'

class BattleField
  module Ship
    SIZE = 10
    X = %w[А Б В Г Д Е Ж З И К]
  end
end
```

В приведенном примере класс `BattleFiled` открывается повторно, и в него добавляется новый модуль `Ship`.

Существует и альтернативная форма объявления вложенного модуля (листинг 19.8).

Листинг 19.8. Альтернативное объявление `BattleField::Ship`. Файл battle_filed_ship.rb

```
require_relative 'battle_field'

module BattleField::Ship
  SIZE = 10
  X = %w[А Б В Г Д Е Ж З И К]
end
```

Вариант, приведенный в листинге 19.8, как правило, более предпочтителен. Такая форма выглядит компактнее и позволяет уменьшить количество отступов.

В листинге 19.9 приводится пример использования класса `BattleFiled` и модуля `BattleFiled::Ship` для создания игрового поля игры «Морской бой».

Листинг 19.8. Игровое поле для «Морского боя». Файл battle_filed_ship_use.rb

```
require_relative 'battle_filed_ship'

fields = BattleField.new(size: BattleField::Ship::SIZE) do |size|
  Array.new(size) do |y|
    Array.new(size) do |x|
      BattleField::Field.new(x: BattleField::Ship::X[x], y: y + 1)
    end
  end
end

puts fields
```

Для того чтобы переопределить игровое поле с участием модуля `BattleFiled::Ship`, используется блок метода `BattleField#initialize`.

Результатом выполнения программы будет следующее игровое поле:

```
А:1 Б:1 В:1 Г:1 Д:1 Е:1 Ж:1 З:1 И:1 К:1
А:2 Б:2 В:2 Г:2 Д:2 Е:2 Ж:2 З:2 И:2 К:2
А:3 Б:3 В:3 Г:3 Д:3 Е:3 Ж:3 З:3 И:3 К:3
А:4 Б:4 В:4 Г:4 Д:4 Е:4 Ж:4 З:4 И:4 К:4
А:5 Б:5 В:5 Г:5 Д:5 Е:5 Ж:5 З:5 И:5 К:5
А:6 Б:6 В:6 Г:6 Д:6 Е:6 Ж:6 З:6 И:6 К:6
А:7 Б:7 В:7 Г:7 Д:7 Е:7 Ж:7 З:7 И:7 К:7
А:8 Б:8 В:8 Г:8 Д:8 Е:8 Ж:8 З:8 И:8 К:8
А:9 Б:9 В:9 Г:9 Д:9 Е:9 Ж:9 З:9 И:9 К:9
А:10 Б:10 В:10 Г:10 Д:10 Е:10 Ж:10 З:10 И:10 К:10
```

Как можно видеть, структура игрового поля немного искажена за счет того, что число 10 содержит две цифры, а числа, меньше 10, — только одну. Исправить ситуацию можно, открыв класс `BattleField::Field` и переопределив метод `to_s` (листинг 19.9).

Листинг 19.9. Переопределение метода `to_s`. Файл battle_filed_ship_improve.rb

```
require_relative 'battle_filed_ship'

class BattleField::Field
  def to_s
    format("%4s", "#{x}:#{y}")
  end
end

fields = BattleField.new(size: BattleField::Ship::SIZE) do |size|
  Array.new(size) do |y|
    Array.new(size) do |x|
      BattleField::Field.new(x: BattleField::Ship::X[x], y: y + 1)
    end
  end
end

puts fields
```

Здесь метод `to_s` исправляется таким образом, чтобы под каждое строковое представление клетки игрового поля отводилось четыре символа. Чтобы добиться этого, строковое представление форматируется методом `format` с использованием шаблона `"%4s"`. Для строк вида `"A:10"` требуется как раз четыре символа, в случае трехсимвольных строк `"A:5"` в их начало добавляется пробел.

Благодаря дополнительному форматированию, каждая клетка поля располагается друг под другом, не выбиваясь за пределы столбца:

```
А:1  Б:1  В:1  Г:1  Д:1  Е:1  Ж:1  З:1  И:1  К:1
А:2  Б:2  В:2  Г:2  Д:2  Е:2  Ж:2  З:2  И:2  К:2
А:3  Б:3  В:3  Г:3  Д:3  Е:3  Ж:3  З:3  И:3  К:3
А:4  Б:4  В:4  Г:4  Д:4  Е:4  Ж:4  З:4  И:4  К:4
А:5  Б:5  В:5  Г:5  Д:5  Е:5  Ж:5  З:5  И:5  К:5
А:6  Б:6  В:6  Г:6  Д:6  Е:6  Ж:6  З:6  И:6  К:6
А:7  Б:7  В:7  Г:7  Д:7  Е:7  Ж:7  З:7  И:7  К:7
А:8  Б:8  В:8  Г:8  Д:8  Е:8  Ж:8  З:8  И:8  К:8
А:9  Б:9  В:9  Г:9  Д:9  Е:9  Ж:9  З:9  И:9  К:9
А:10 Б:10 В:10 Г:10 Д:10 Е:10 Ж:10 З:10 И:10 К:10
```

В больших программах вложенные модули редко располагаются в одном файле с классом. Если для имени классов и модулей используется CamelCase-стиль, для названий файлов всегда используется snake-стиль (см. *разд. 2.3.2*). Последнее связано с тем, что не все файловые системы различают большие и маленькие буквы в именах файлов.

Как правило, основной класс или модуль располагается в одноименном Ruby-файле. Вложенные классы и модули располагаются в каталоге, название которого совпадает с именем Ruby-файла. Под каждый вложенный модуль или класс в каталоге выделяют отдельный файл (рис. 19.2). При такой организации проекта можно легко найти класс самостоятельно, кроме того, значительно облегчается построение систем автозагрузки классов.

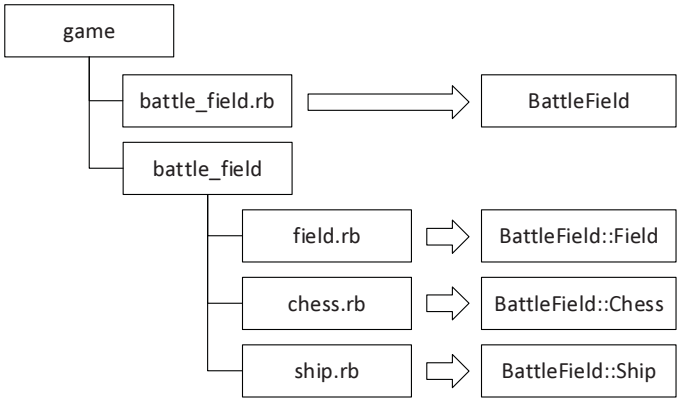


Рис. 19.2. Структура вложенных классов и модулей

Представленная здесь система расположения и именования вложенных модулей является обязательной в большинстве Ruby-фреймворков, включая самый популярный — Ruby on Rails.

19.5. Доступ к глобальным классам и модулям

Имена классов и модулей — это обычные константы языка Ruby. Как нам уже известно из *главы 6*, попытка их переопределения приводит к выдаче предупреждения. Вместо этого мы используем вложенные классы и модули, чтобы изолировать константы друг от друга.

Такая техника имеет обратную сторону медали — вложенные классы и модули блокируют доступ к одноименным глобальным (стандартным) классам и модулям (рис. 19.3).

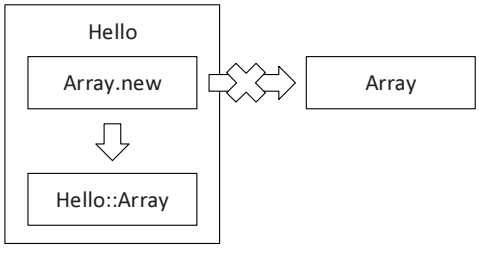


Рис. 19.3. Вложенные модули и классы блокируют доступ к стандартным классам и модулям

Рассмотрим проблему на примере вложенного класса `Array`. В листинге 19.10 приводится пример класса `Hello`, в который вкладывается класс `Array`. Вложенный класс `Array` унаследован от класса `Object` и не имеет никакого отношения к стандартным Ruby-массивам.

> ***Замечание***
>
> В примерах этого раздела мы используем вместо модулей классы. В следующей главе мы утвердимся во мнении, что классы и модули — это близкие родственники. Так что все, что мы описываем здесь для классов, будет справедливо и для модулей. Однако модулями воспользоваться сложнее, и, чтобы не забегать вперед, вместо них мы здесь используем классы, с которыми же успели познакомиться.

Листинг 19.10. Экранирование стандартного класса. Файл array_wrong.rb

```
class Hello
  attr_accessor :list

  def initialize
    @list = Array.new
  end

  class Array
  end
end

hello = Hello.new
hello.list << 'ruby' # undefined method `<<'
```

В методе `Hello#initialize` инстанс-переменной `@list` присваивается объект `Array.new`. Это объект класса `Hello::Array`, а не стандартного класса массива `Array`. Поэтому попытка использовать оператор `<<` для его заполнения заканчивается неудачей.

Можно воспроизвести ситуацию в консоли интерактивного Ruby (`irb`), и мы здесь ожидаем стандартного поведения класса Array:

```
 > list = Array.new
=> []
 > list << 'ruby'
=> ["ruby"]
```

Вместо этого внутри класса `Hello` происходит создание объекта `Hello::Array`:

```
 > require_relative 'array_wrong'
 > list = Hello::Array.new
=> #<Hello::Array:0x00007fe0a182a2d8>
 > list << 'ruby'
NoMethodError (undefined method `<<')
```

Исправить ситуацию можно, воспользовавшись оператором разрешения области видимости. У глобальной области нет имени, поэтому слева от оператора просто ничего не указывается:

```
 > list = ::Array.new
=> []
 > list << 'ruby'
=> ["ruby"]
```

Перепишем класс `Hello`, чтобы в его инстанс-переменной `@list` присваивался массив, а не объект `Hello::Array` (листинг 19.11).

Листинг 19.11. Доступ к массивам `::Array`. Файл array.rb

```
class Hello
  attr_accessor :list

  def initialize
    @list = ::Array.new
  end

  class Array
  end
end

hello = Hello.new
hello.list << 'ruby'
p hello.list # ["ruby"]
```

Получившийся здесь пример намеренно немного искусственный. Его задача — продемонстрировать проблему экранирования констант. Однако такая форма обращения к классам и модулям глобальной области видимости весьма распространена.

В листинге 19.12 приводится пример класса палитры `Pallete`, в объект которой можно добавлять массив цветов. В нашем случае добавляется семь цветов радуги.

Листинг 19.12. Класс палитры `Pallete`. Файл pallete.rb

```
class Pallete
  def initialize(colors)
    @param = Array.new
    colors.each do |color|
      @param.set(color)
    end
  end

  def report
    @param.each do |color|
      puts color
    end
  end

  class Array
    def initialize
      @arr = ::Array.new
    end
    def set(value)
      @arr << value
    end
    def each
      @arr.each do |element|
        yield element
      end
    end
  end
end

colors = %i[red orange yellow green blue indigo violet]
pallete = Pallete.new(colors)
pallete.report
```

Для хранения цветов внутри класса `Pallete` используется вложенный класс `Pallete::Array`. Этот класс сам является оберткой вокруг стандартного класса массива `Array`. Чтобы до него добраться, используется оператор разрешения области видимости `::Array`. Результатом работы программы будет список цветов радуги:

```
red
orange
yellow
green
blue
indigo
violet
```

Вместо `::Array` можно было так же воспользоваться синтаксическим конструктором «квадратные скобки» `[]` (листинг 19.13).

Листинг 19.13. Альтернативная реализация `Pallete`. Файл pallete_alternative.rb

```
...
  class Array
    def initialize
      @arr = []
    end
...
```

Задания

1. Создайте класс `Vector`, моделирующий вектор в двумерном пространстве координат. Объекты класса должны позволять задать две точки координат: начало и конец вектора. Для задания свойств точки создайте вложенный класс `Vector::Point`. Реализуйте метод `distance`, позволяющий вычислить длину вектора.
2. Создайте класс `Unit`, моделирующий коллектив из семи человек: руководитель, два бэкенд-разработчика, два фронтенд-разработчика, тестировщик и дизайнер. Для моделирования каждого сотрудника создайте вложенный класс `Unit::Employee`. Объект класса `Unit` должен позволять добавлять, удалять, редактировать сотрудников и выводить их полный список в алфавитном порядке. Кроме того, создайте метод или набор методов, позволяющих фильтровать команду по ролям — например, запрашивать список тестировщиков или бэкенд-разработчиков.

ГЛАВА 20

Подмешивание модулей

Файлы с исходными кодами этой главы находятся в каталоге *mixins* сопровождающего книгу электронного архива.

Модули выступают контейнерами методов и могут быть включены в класс или объект, расширяя их своими методами. Тем самым компенсируется отсутствие в Ruby множественного наследования. При помощи модулей можно обеспечить множество не связанных друг с другом объектов и классов одинаковым поведением.

Такое свойство модулей используется не только в пользовательских программах. Стандартные модули `Kernel`, `Comparable` и `Enumerable` играют ключевую роль в объектно-ориентированной схеме Ruby. Изучение модулей крайне важно для понимания механизма поиска методов в Ruby, а также для создания компактного и элегантного кода.

С другой стороны, модули — это объекты класса `Module`, который является базовым классом для класса `Class`. Таким образом, классы — это тоже модули, которые расширены возможностью создавать собственные объекты и наследовать свойства друг друга. По сравнению с классами, модули более простые — нельзя создать объект модуля, невозможно наследовать модули друг от друга.

В Ruby-сообществе принято использовать классы везде, где требуются объекты или отношение наследования. Во всех остальных случаях задействуются модули.

20.1. Класс *Module*

Как уже известно из *главы 19*, для создания простейшего модуля следует воспользоваться ключевым словом `module` (листинг 20.1).

Листинг 20.1. Создание модуля. Файл module.rb

```
module Site
end
```

Модуль `Site` является объектом. Мы можем загрузить его в консоль интерактивного Ruby (`irb`) и убедиться в этом на практике:

```
 > require_relative 'module'
=> true
 > Site.class
=> Module
```

Как можно видеть, модуль — это объект класса `Module`. Так как модули и классы очень похожи, рассмотрим их в сравнении друг с другом.

Классы всегда зависят от своего базового класса, исключение составляет лишь класс `BasicObject`, который не имеет предка. В модуле не реализован метод `superclass` или его аналог — он просто не нужен:

```
 > Object.superclass
=> BasicObject
 > Site.superclass
NoMethodError (undefined method `superclass' for Site:Module)
```

Соответственно, модули не связаны наследованием — они, как и любые другие объекты, существуют сами по себе (рис. 20.1).

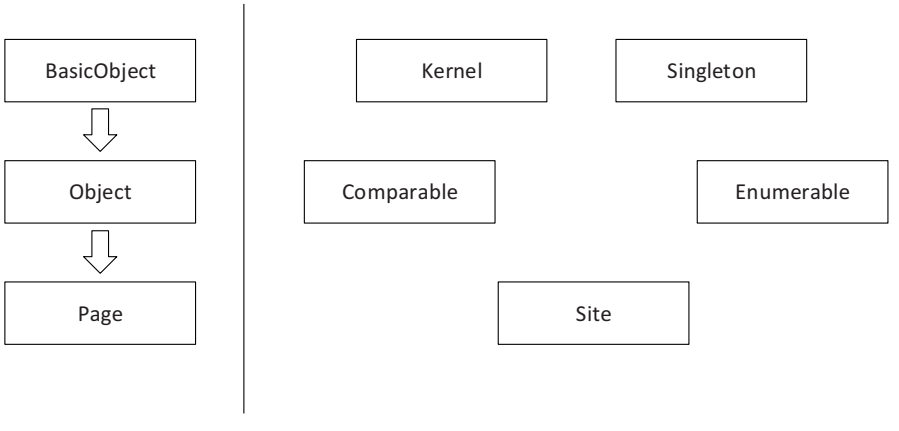


Рис. 20.1. Классы всегда увязаны в иерархию наследования, модули существуют сами по себе

Тем не менее, как и любой объект языка Ruby, модуль связан с двумя классами: классом `Module`, который определяет свойства объектов-модулей, и метаклассом, в который помещаются синглетон-методы, присущие только текущему объекту:

```
 > Class.superclass
=> Module
 > Module.superclass
=> Object
```

То есть классы и модули — это близкие родственники. Класс `Module` является базовым для класса `Class`. Класс — это тот же модуль, только снабженный возможностями для создания объектов и наследования.

Используя новые сведения, расширим объектно-ориентированную схему языка Ruby. В вершине иерархии находится класс `BasicObject`, от которого наследуется класс `Object`. От класса `Object` наследует большинство классов языка Ruby, включая класс `Module`. От класса `Module` наследуется класс `Class` (рис. 20.2).

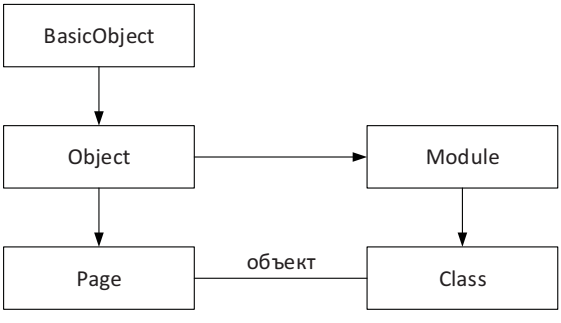


Рис. 20.2. Связь модулей с классами

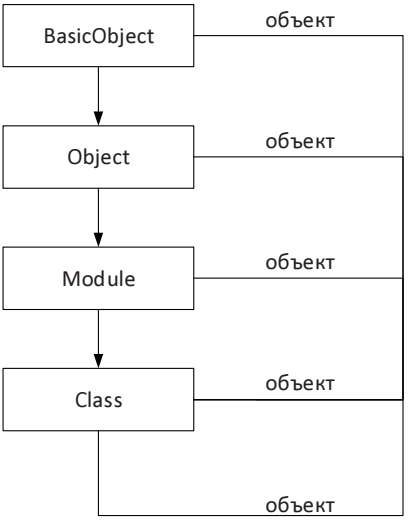


Рис. 20.3. Все классы являются объектами класса `Class`

Самое удивительное, что все пять классов на рис. 20.2 являются объектами класса `Class` (рис. 20.3). Это одна из самых путающих частей языка — указанное противоречие разрешено на уровне интерпретатора Ruby при помощи кода на языке Си.

Таким образом, модули, в отличие от классов, — это просто контейнеры для констант и методов.

20.2. Подмешивание модулей в класс

В *разд. 19.2* мы столкнулись с тем, что воспользоваться методом модуля не просто, т. к. у модуля нет объектов. Один из способов добраться до методов модуля — подмешать модуль в класс или объект, чтобы методы модуля стали методами объекта.

В Ruby не поддерживается множественное наследование, как в языках программирования Python или C++, и у каждого класса может быть только один базовый класс. Такая схема наследования позволяет строить иерархические деревья, не допуская графов (рис. 20.4).

Тем не менее, в мире не все задачи описываются при помощи иерархии и деревьев. Более того, круг таких задач ограничен. Именно поэтому в объектно-ориентированном программировании чаще прибегают к композиции, чем к наследованию.

То есть на практике гораздо чаще встречается ситуация, когда объект состоит из других объектов. Например, автомобиль состоит из четырех колес, кузова, двигателя. Гораздо реже встречаются иерархии, где наследование может быть полезным. Например, объект => фигура => треугольник => равнобедренный треугольник => равносторонний треугольник.

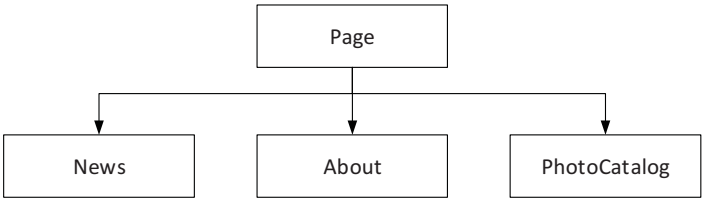


Рис. 20.4. Классы в Ruby допускают только одиночное наследование

Рассмотрим ограничение наследования на примере небольшой иерархии классов, моделирующей страницы сайта (листинг 20.2).

Листинг 20.2. Иерархия страниц сайта. Файл page.rb

```
class Page
  attr_accessor :title, :body
end

class News < Page
  attr_accessor :date
end

class About < Page
  attr_accessor :phones, :address
end

class PhotoCatalog < Page
  attr_accessor :photos
end
```

Базовый класс `Page` задает общие свойства для всех страниц:

- `title` — название страницы;
- `body` — содержимое страницы.

От базового класса унаследовано три производных: `News` — новости, `About` — страница «О нас» и `PhotoCatalog` — фотогалерея. Каждый из классов расширяет исходный набор собственными атрибутами. В новости добавляется дата публикации `:date`, в класс `About` добавляются список телефонов `phones` и адрес `:address`, фотогалерея расширяется списком фотографий `:photos`.

Предположим, что спустя какое-то время необходимо снабдить часть страниц дополнительными атрибутами. Допустим, в качестве таких атрибутов выступает

SEO-информация, предназначенная главным образом для роботов поисковых систем. Эти атрибуты не видны посетителям сайта и размещаются в специализированных HTML-тэгах. В этот набор атрибутов могут входить:

- `meta_title` — заголовок для тэга `<title>`;
- `meta_description` — описание сайта для meta-тэга `description`;
- `meta_keywords` — ключевые слова для meta-тэга `keywords`.

На странице фотогалереи `PhotoCatalog` практически нет никакой текстовой информации. Поэтому SEO-информация должна попасть только в классы `News` и `About` (рис. 20.5).

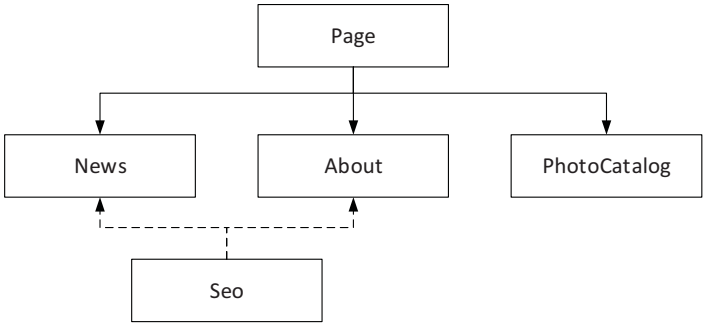


Рис. 20.5. Добавление SEO-информации на уровень классов `News` и `About`

Если поместить SEO-атрибуты на уровень базового класса `Page`, то в результате наследования они волей-неволей попадут в класс `PhotoCatalog`.

Можно скопировать SEO-атрибуты и в класс `News`, и в класс `About`. Однако в этом случае будет нарушен принцип DRY (Don't repeat yourself, не повторяйся) — если впоследствии потребуется изменить набор параметров, их придется менять в нескольких местах. Рано или поздно мы забудем где-нибудь поправить код, и возникнет ошибка. Важно добиться ситуации, когда исправление в коде нужно было бы вносить лишь в одном месте.

Здесь пригодилось бы множественное наследование, когда классы `News` и `About` можно было бы унаследовать от еще одного класса. Язык Ruby предоставляет альтернативный механизм для решения этой задачи. SEO-атрибуты можно выделить в модуль и подмешать в классы.

В листинге 20.3 представлен модуль `Seo`, который при помощи `attr_accessor` объявляет геттеры и сеттеры для `meta_title`, `meta_description` и `meta_keywords`.

> ***Замечание***
>
> Префикс `meta_` в атрибутах `meta_title`, `meta_description` и `meta_keywords` является признаком непродуманной структуры. Как правило, это означает, что в коде не хватает либо класса, либо объекта с атрибутами `title`, `description` и `keywords`. В качестве упражнения избавьтесь от этого префикса.

Листинг 20.3. Модуль Seo. Файл seo_module.rb

```
module Seo
  attr_accessor :meta_title, :meta_description, :meta_keywords
end
```

Теперь можно подмешать модуль Seo только в классы News и About. Для этого используется метод include (листинг 20.4).

> ***Замечание***
> На профессиональном сленге подмешиваемые модули называют *примесями*, или *миксинами* (Mixins).

Листинг 20.4. Подмешивание модуля Seo при помощи include. Файл page_include.rb

```
require_relative 'seo_module'

class Page
  attr_accessor :title, :body
end

class News < Page
  include Seo
  attr_accessor :date
end

class About < Page
  include Seo
  attr_accessor :phones, :address
end

class PhotoCatalog < Page
  attr_accessor :photos
end
```

В листинге 20.5 создается объект класса About, в котором используются сеттеры и геттеры из модуля Seo.

Листинг 20.5. Создание объекта класса About. Файл about.rb

```
require_relative 'page_include'

about = About.new

about.title = 'О нас'
about.body = 'Вы сможете обнаружить нас по адресам'
about.phones = ['+7 920 4567722', '+7 920 4567733']
about.address = '191036, Санкт-Петербург, ул. Гончарная, дом 20, пом. 7Н'
```

```
about.meta_title = about.title
about.meta_description = "Адрес: #{about.address}"
about.meta_keywords = ['О нас', 'Адреса', 'Телефоны']

p about.title
p about.body
p about.phones.join ', '
p about.address

p about.meta_title
p about.meta_description
p about.meta_keywords.join ', '
```

Как можно видеть, благодаря подмешиванию, объекты класса `About` и `News` получают гетеры и сеттеры, определенные в модуле `Seo`. Причем методы появляются только в этих классах, классы `Page` и `PhotoCatalog` указанных методов не содержат (листинг 20.6).

Листинг 20.6. Файл photo_catalog.rb

```
require_relative 'page_include'

photos = PhotoCatalog.new

p photos.respond_to? :title            # true
p photos.respond_to? :body             # true

p photos.respond_to? :meta_title       # false
p photos.respond_to? :meta_description # false
p photos.respond_to? :meta_keywords    # false
```

20.3. Подмешивание модулей в объект

Модули можно подмешивать не только в класс, но и на уровень объекта. Только в этом случае вместо метода `include` потребуется метод `extend` (листинг 20.7).

Листинг 20.7. Подмешивание модуля при помощи `extend`. Файл extend.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
end

ticket = Object.new
```

```
ticket.extend Hello

puts ticket.say('Ruby') # Hello, Ruby!
```

В приведенном примере создается объект `ticket`, возможности которого расширяются модулем `Hello`. Для этого модуль `Hello` передается методу `extend`. После этого метод `Hello#say` становится инстанс-методом объекта.

В глобальной области мы уже находимся в рамках `main`-объекта:

```
 > self
=> main
 > self.class
=> Object
```

Этот объект можно расширять модулем так же, как и любой другой объект языка Ruby. Метод `extend` можно вызывать без получателя, поскольку получателем по умолчанию выступает `self`, который ссылается на объект глобальной области видимости. Впрочем, явное указание получателя `self.extend` тоже не будет ошибкой. В качестве аргумента методу передается модуль (листинг 20.8).

Листинг 20.8. Подмешивание модуля при помощи `extend`. Файл main_extend.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
end

extend Hello

puts say('Ruby') # Hello, Ruby!
```

В результате методы модуля `Hello` становятся глобальными методами.

Так как методы класса — это синглетон-методы объекта класса, при помощи `extend` можно создавать методы класса. В листинге 20.9 создается класс `Greet`, который при помощи модуля `extend` расширяется методом `Hello`. Это приводит к созданию метода `Geet::say`.

Листинг 20.9. Создание метода класса. Файл class_method.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
end
```

```
class Greet
  extend Hello
end

puts Greet.say('Ruby') # Hello, Ruby!
```

Таким образом, подмешивание модуля в класс при помощи метода `include` создает инстанс-методы, которые можно вызывать у объекта класса. Использование метода `extend` позволяет создать методы класса, для вызова которых не нужно создание объекта — в качестве получателя метода выступает класс (рис. 20.6).

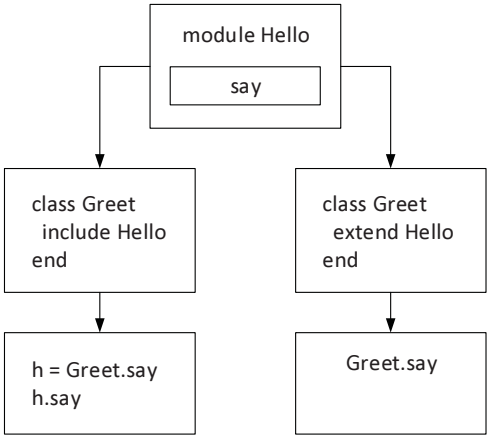


Рис. 20.6. Разница между подмешиванием модулей при помощи `include` и `extend`

Для подключения инстанс-методов и методов класса используются разные способы: `include` и `extend`. Поэтому часто методы, предназначенные для этих двух ролей, разделяют по разным модулям. Более того, один из таких модулей можно сделать вложенным (рис. 20.7).

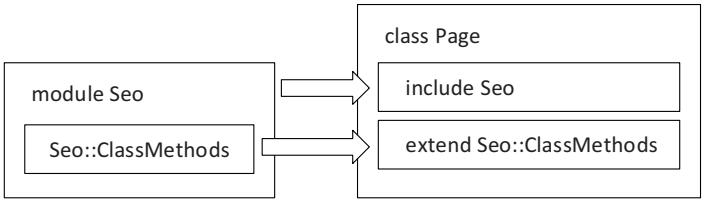


Рис. 20.7. Вложенный модуль `Seo::ClassMethods`

Продемонстрируем использование вложенного модуля для методов класса на практике (листинг 20.10).

Листинг 20.10. Выделение методов класса во вложенный модуль. Файл seo.rb

```
module Seo
  attr_accessor :meta_title, :meta_description, :meta_keywords
```

```
  module ClassMethods
    def title(name)
      "Программирование на языке Ruby. #{name}."
    end
    def say(name)
      "Hello, #{name}!"
    end
  end
end
```

В приведенном примере показан модуль `Seo`, в котором методы `meta_title`, `meta_title=`, `meta_description`, `meta_description=`, `meta_keywords` и `meta_keywords=` предназначены для подключения через `include`. Эти шесть методов создаются при помощи `attr_accessor`. Им предстоит стать инстанс-методами класса, в который будет подмешан модуль `Seo`.

Методы класса `title` и `hello` выделены в отдельный подмодуль `ClassMethods`. Этот вложенный модуль подключается при помощи `extend`. Методы `title` и `hello` станут будущими методами класса.

В листинге 20.11 приводится пример подключения модуля `Seo` в класс `Page`. Так как модуль `ClassMethods` является вложенным, чтобы до него добраться, необходимо использовать полное имя `Seo::ClassMethods`.

Листинг 20.11. Подключение модуля `Seo`. Файл `seo_use.rb`

```
require_relative 'seo'

class Page
  include Seo
  extend Seo::ClassMethods
end

p Page.instance_methods
# [:meta_description=, :meta_title, :meta_description, ...
p Page.methods
# [:title, :say, ...
```

Несмотря на то, что классы наследуется от модулей, использовать их для подмешивания запрещается (листинг 20.12).

Листинг 20.12. Запрещено подмешивание классов. Файл `class_as_module.rb`

```
class Hello
  def say(str)
    "Hello, #{str}!"
  end
end
```

```
class Greet
  include Hello # wrong argument type Class (expected Module) (TypeError)
end

obj = Greet.new

obj.extend Hello # wrong argument type Class (expected Module) (TypeError)
```

20.4. Синглетон-методы модуля

Так как модуль является объектом, для него можно создать синглетон-методы (листинг 20.13).

Листинг 20.13. Синглетон-метод `say` модуля `Hello`. Файл hello.rb

```
module Hello
  def self.say(name)
    "Hello, #{name}!"
  end
end
```

В теле модуля ключевое слово `self` ссылается на сам модуль `Hello`. Поэтому выражение `self.say` аналогично `Hello.say`. Программу можно также переписать при помощи конструкции `class << self` (листинг 20.14).

Листинг 20.14. Использование конструкции `class << self`. Файл hello_class_self.rb

```
module Hello
  class << self
    def say(name)
      "Hello, #{name}!"
    end
  end
end
```

Вариант с `class << self` обычно используют, если в модуль необходимо добавить несколько синглетон-методов. Если синглетон-метод один, предпочитают вариант из листинга 20.13.

Синглетон-методы модулей можно вызывать напрямую, используя в качестве получателя модуль (листинг 20.15).

Листинг 20.15. Вызов синглетон-метода `Hello::say`. Файл hello_use.rb

```
require_relative 'hello'
puts Hello.say('Ruby') # Hello, Ruby!
```

Обычные методы модуля можно превратить в синглетон-методы, если применить метод `extend` в отношении ключевого слова `self` (листинг 20.16).

Листинг 20.16. Превращение обычных методов в методы класса. Файл extend_self.rb

```
module Hello
  extend self

  def say(name)
    "Hello, #{name}!"
  end
end

puts Hello.say 'Ruby' # Hello, Ruby!
```

Модуль `Hello` из приведенного примера имеет два метода: `say` и `self.say`.

Хотя запись из листинга 20.16 вполне допустима, она требует от разработчика значительных усилий для ее расшифровки: необходимо вспомнить свойства модулей, как работает метод `extend`, на что ссылается ключевое слово `self`, и какие методы включены в настоящий момент в модуль. Кроме того, не часто требуется превращать все инстанс-методы модуля в синглетон-методы.

Поэтому Ruby предоставляет более удобный способ перевода обычного метода модуля в синглетон-метод. Для этого используется метод `module_function`, который в качестве аргументов принимает список символов-имен. Эти методы становятся доступными и в качестве синглетон-методов модуля, и в качестве инстанс-методов будущего класса (рис. 20.8).

В листинге 20.17 приводится пример использования метода `module_function` для превращения инстанс-метода `say` модуля `Hello` в метод модуля.

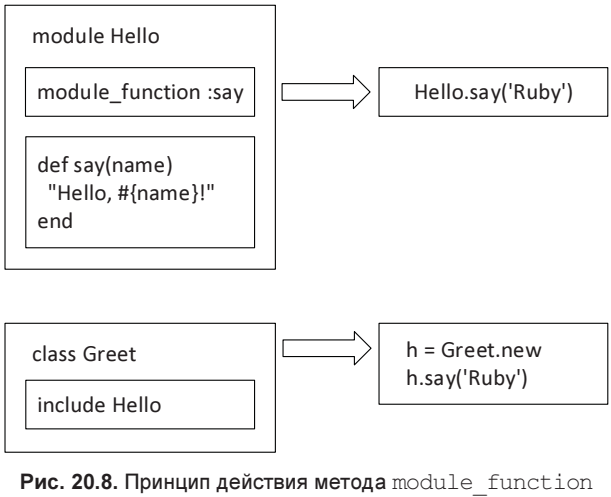


Рис. 20.8. Принцип действия метода `module_function`

Листинг 20.17. Использование метода `module_function`. Файл module_function.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end

  module_function :say
end

puts Hello.say('Ruby') # Hello, Ruby!
```

Следует иметь в виду, что метод `module_function` имеет побочное действие. Инстанс-методы, которые передаются ему в качестве аргумента, становятся закрытыми. Их можно успешно использовать внутри класса, однако они не доступны для вызова в отношении объектов классов (листинг 20.18).

Листинг 20.18. Файл module_function_private.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end

  module_function :say
end

class Greet
  include Hello

  def hello_world
    say('world')
  end
end

g = Greet.new
puts g.hello_world # Hello, world!
puts g.say('Ruby') # private method `say' called
```

Итак, модули могут содержать инстанс- и синглетон-методы. Инстанс-методы можно вызывать на уровне класса, если модуль подмешан с помощью `extend`, или их можно вызывать на уровне объекта класса, если модуль подмешан с помощью `include` (рис. 20.9).

Синглетон-методы можно вызывать непосредственно на уровне модуля. Подмешивая исходный модуль в класс или модуль, нет возможности ими воспользоваться. Синглетон-методы — это методы одного объекта, в данном случае — модуля.

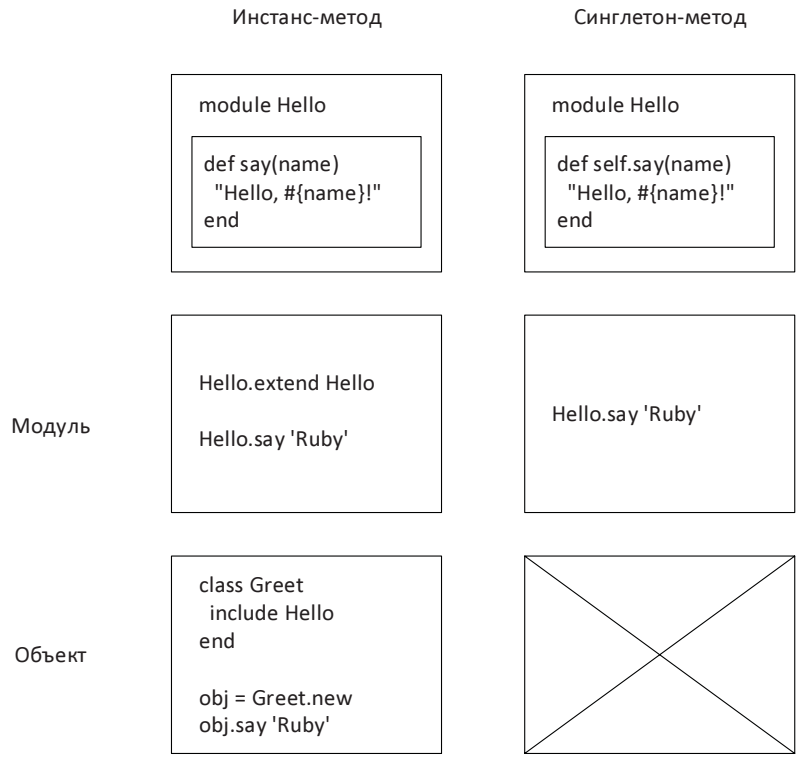


Рис. 20.9. Инстанс- и синглетон-методы модуля

20.5. Области видимости

Класс `Class` расширяет `Module` лишь четырьмя дополнительными методами: `new`, `allocated`, `inherited` и `superclass`. Все остальные методы реализованы на уровне класса `Module`.

Это означает, что методы управления областями видимости `public`, `private` и `protected` определены на уровне модуля и могут быть в нем задействованы (листинг 20.19).

Замечание

Области видимости детально рассматривались в *главе 18*.

Листинг 20.19. Области видимости в модуле. Файл scope.rb

```
module Scope
  public

  def say(name)
    "Scope#say: Hello, #{name}!"
  end
```

```
  protected

  def greeting
    "Scope#greeting: Hello, world!"
  end

  private

  def hello
    "Scope#hello: Hello, world!"
  end
end
```

В модуле `Scope` определены три метода: открытый метод `say`, защищенный метод `greeting` и закрытый метод `hello`. Можно было не использовать явно метод `public`, т. к. все методы в модуле по умолчанию являются открытыми.

При подмешивании модуля `Scope` в класс методы сохраняют свои области видимости (листинг 20.20).

Листинг 20.20. Подмешивание модуля `Scope`. Файл scope_include.rb

```
require_relative 'scope'

class HelloWorld
  include Scope
end

h = HelloWorld.new

puts h.say('Ruby') # Scope#say: Hello, Ruby!
puts h.greeting    # protected method `greeting' called for
puts h.hello       # private method `hello' called for
```

За пределами класса `HelloWorld` можно обратиться только к методу `say`. Методы `greeting` и `hello` доступны лишь внутри класса, в который подмешен модуль `Scope`.

Области видимости методов остаются в силе и при подмешивании модуля при помощи `extend` (листинг 20.21).

Листинг 20.21. Подмешивание модуля методом `extend`. Файл scope_extend.rb

```
require_relative 'scope'

class HelloWorld
  extend Scope
end
```

```
puts HelloWorld.say('Ruby') # Scope#say: Hello, Ruby!
puts HelloWorld.greeting    # protected method `greeting' called for
puts HelloWorld.hello       # private method `hello' called for
```

При помощи методов `public`, `protected` и `private` можно управлять областями видимости синглетон-методов модуля (листинг 20.22).

Листинг 20.22. Файл scope_class.rb

```
module Scope
  class << self
    def say(name)
      "Scope::say: Hello, #{name}!"
    end

    def get_greeting
      self.greeting
    end

    def get_hello
      hello
    end

    protected

    def greeting
      "Scope::greeting: Hello, world!"
    end

    private

    def hello
      "Scope::hello: Hello, world!"
    end
  end
end

puts Scope.say('Ruby')  # Scope::say: Hello, Ruby!
puts Scope.get_greeting # Scope::greeting: Hello, world!
puts Scope.get_hello    # Scope::hello: Hello, world!

puts Scope.greeting  # protected method `greeting' called for Scope:Module
puts Scope.hello     # private method `hello' called for Scope:Module
```

В приведенном примере в модуле `Scope` объявляется пять синглетон-методов. Для этого при помощи конструкции `class << self` открывается метакласс модуля. Метод `hello` является закрытым, `greeting` — защищенным. Методы `say`, `get_greeting` и `get_hello` являются открытыми. Методы `hello` и `greeting` нельзя вызывать за пре-

делами модуля, зато их можно вызывать внутри других синглетон-методов модуля `Scope`.

Если синглетон-методы создаются при помощи префикса `self.`, воспользоваться `public`, `private` и `protected` не получится. В этом случае их действие будет распространяться лишь на инстанс-методы модуля.

Закрыть синглетон-методы можно альтернативным способом, воспользовавшись методом `private_class_method` (листинг 20.23).

Листинг 20.23. Использование метода `private_class_method`. Файл scope_singleton.rb

```
module Scope
  def self.say(name)
    "Scope::say: Hello, #{name}!"
  end

  def self.get_greeting
    greeting
  end

  def self.get_hello
    hello
  end

  def self.greeting
    "Scope::greeting: Hello, world!"
  end

  def self.hello
    "Scope::hello: Hello, world!"
  end

  private_class_method :hello, :greeting
end

puts Scope.say('Ruby')  # Scope::say: Hello, Ruby!
puts Scope.get_greeting # Scope::greeting: Hello, world!
puts Scope.get_hello    # Scope::hello: Hello, world!

puts Scope.greeting  # private method `greeting' called for Scope:Module
puts Scope.hello     # private method `hello' called for Scope:Module
```

20.6. Стандартный модуль *Kernel*

Ruby предоставляет ряд стандартных модулей. Одним из самых важных является модуль `Kernel`. Точно так же, как мы подмешивали свои собственные модули в классы, `Kernel` подмешан в стандартный класс `Object`. В этом модуле сосредото-

чены все методы из глобальной области видимости. Благодаря тому, что `Object` — это класс глобальной области видимости, и от него наследуются все остальные классы языка Ruby (кроме `BasicObject`), методы `Kernel` доступны в любой точке Ruby-программы.

> ***Замечание***
>
> `Kernel` не единственный модуль, который предоставляет язык программирования Ruby. Стандартным модулям посвящена *глава 21*.

В листинге 20.24 приводятся два вызова глобального метода `puts`: один вызов привычный — без получателя, другой — как синглетон-метод модуля `Kernel`.

Листинг 20.24. Альтернативные способы вызова глобальных методов. Файл puts.rb

```
puts 'Hello, world!'        # Hello, world!
Kernel.puts 'Hello, world!' # Hello, world!
```

На практике предпочитают более короткий вариант. Однако важно помнить, в каком модуле сосредоточены глобальные методы.

Модули можно открыть точно так же, как классы (см. *разд. 13.5*). Вместо прямого определения метода, можно открыть модуль `Kernel` и добавить метод в него (листинг 20.25).

Листинг 20.25. Добавление метода в модуль `Kernel`. Файл say_kernel.rb

```
module Kernel
  def say
    'hello'
  end
end

puts say # hello
```

Метод `say` становится доступным, как обычный глобальный метод. На практике нет необходимости открывать модуль `Kernel` и добавлять в него методы. Более правильно просто создать обычный метод в глобальной области (листинг 20.26).

Листинг 20.26. Добавление метода в модуль `Kernel`. Файл say.rb

```
def say
  'hello'
end

puts say # hello
```

Модуль `Kernel` встраивается в цепочку наследования стандартных классов Ruby (рис. 20.10). Это означает, что внутри класса или объекта всегда можно переопределить любой глобальный метод. Например, по умолчанию методы `Kernel` объяв-

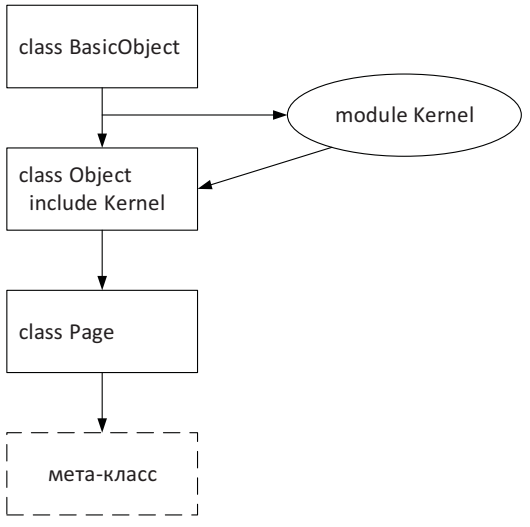


Рис. 20.10. Модуль `Kernel` подмешивается на уровень класса `Object`

лены как закрытые. Именно поэтому невозможно использовать глобальные методы с каким-либо объектом-получателем. Исключение составляет сам метод `Kernel`, т. к. глобальные методы дополнительно объявлены как синглетон-методы модуля:

```
 > self.puts 'Hello, world!'
NoMethodError (private method `puts' called for main:Object)
 > Object.puts 'Hello, world!'
NoMethodError (private method `puts' called for Object:Class)
```

Как видно из сообщений об ошибках, метод `puts` имеется и у объекта глобальной области, и у любого другого объекта языка Ruby. Однако метод объявлен закрытым, и невозможно получить к нему доступ, используя получатель.

Однако в своем собственном классе можно переопределить метод `puts`, объявив его открытым (листинг 20.27).

Листинг 20.27. Переопределение метода `say`. Файл puts_overload.rb

```
class HelloWorld
  def puts(*params)
    super
  end
end

hello = HelloWorld.new
hello.puts 'Hello, world!' # Hello, world!
```

Для доступа к предыдущему объявлению метода `puts` в модуле `Kernel` используется ключевое слово `super` (рис. 20.11). Как видно из приведенного примера, метод `puts` для объектов `hello` доступен как открытый инстанс-метод.

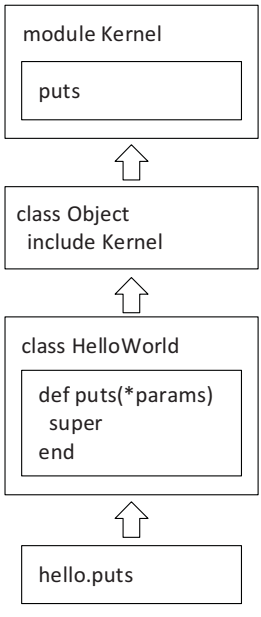


Рис. 20.11. Поиск метода `puts`

Цепочку наследования всегда можно проверить при помощи метода `ancestors`:

```
 > require_relative 'puts_overload'
Hello, world!
=> true
 > HelloWorld.ancestors
=> [HelloWorld, Object, Kernel, BasicObject]
```

Приведенная цепочка показывает список классов и модулей (а также их порядок), в которых будет осуществляться поиск метода. В эту цепочку не включается метакласс объекта, т. к. к нему не привязана константа, — получить ссылку на него можно либо при помощи ключевого слова `self`, либо при помощи метода `singleton_class`.

20.7. Поиск методов в модулях

Поиск метода не ограничивается только модулем `Kernel`. При включении модуля в класс при помощи метода `include` модуль встраивается в цепочку поиска метода (рис. 20.12).

Модуль `Greetable` здесь подмешивается на уровень класса `Hello`, который в свою очередь наследуется от класса `Greet`. Модуль встраивается между базовым и производным классами.

Если метод `say` реализован на уровне обоих классов и модуля, то при его поиске он сначала будет обнаружен в классе `Hello` (листинг 20.28).

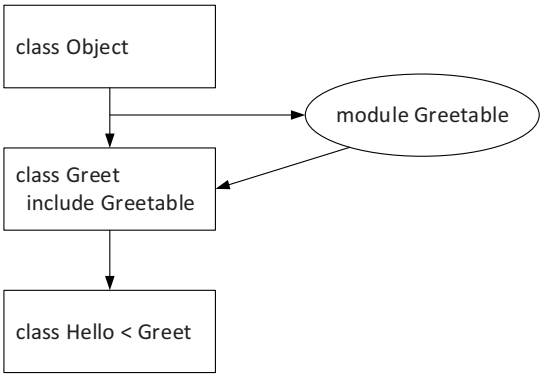


Рис. 20.12. Подмешивание модуля

Листинг 20.28. Поиск методов в классах и модулях. Файл greetable_include.rb

```
class Greet
  def say(name)
    "Greet#say: Hello, #{name}!"
  end
end

module Greetable
  def say(name)
    "Greetable#say: Hello, #{name}!"
  end
end

class Hello < Greet
  include Greetable

  def say(name)
    "Hello#say: Hello, #{name}!"
  end
end

hello = Hello.new
puts hello.say('Ruby') # Hello#say: Hello, Ruby!
```

Если в приведенном примере убрать метод `say` из класса `Hello`, вызов `hello.say` будет вызывать метод из модуля `Greetable`. Если метод убрать из модуля, будет обнаруживаться метод `Greet#say`. Если убрать объявление метода из базового класса `Greet`, то после безуспешного поиска метода в классе `Object`, модуле `Kernel` и классе `BasicObject`, Ruby-интерпретатор выдаст ошибку обращения к несуществующему методу.

Если в класс включается несколько модулей, содержащих один и тот же метод, то используется метод из последнего модуля (листинг 20.29).

Листинг 20.29. Поиск метода в нескольких модулях. Файл few_modules.rb

```
module Seo
  def title
    "Seo#title"
  end
end

module Title
  def title
    "Title#title"
  end
end

class Page
  include Seo
  include Title
end

page = Page.new
puts page.title # Title#title
```

Если возникают сомнения, в каком порядке будет осуществляться поиск метода в сложной цепочке наследования и включений модулей, всегда можно вызвать метод `ancestors` (листинг 20.30).

Листинг 20.30. Поиск метода в нескольких модулях. Файл ancestors.rb

```
require_relative 'greetable_include.rb'
require_relative 'few_modules'

p Hello.ancestors # [Hello, Greetable, Greet, Object, Kernel, BasicObject]
p Page.ancestors  # [Page, Title, Seo, Object, Kernel, BasicObject]
```

Интерпретатор Ruby будет искать методы именно в том порядке, который возвращает `ancestors`.

Модуль можно попытаться подмешать в класс несколько раз. Однако повторное включение модуля просто игнорируется (листинг 20.31).

Листинг 20.31. Попытка повторного включения модуля. Файл include_repeat.rb

```
module Seo
  def title
    "Seo#title"
  end
end
```

```
module Title
  def title
    "Title#title"
  end
end

class Page
  include Seo
  include Title
  include Seo
end

page = Page.new
puts page.title # Title#title
```

Несмотря на то, что модуль `Seo` включается в класс `Page` последним, будет вызван метод `Title#title`. Цепочка поиска, которая возвращается методом `ancestors`, тоже останется неизменной:

```
[Page, Title, Seo, Object, Kernel, BasicObject]
```

Логический метод `include?` позволяет выяснить, входит ли модуль в цепочку поиска. Метод принимает в качестве параметра модуль и возвращает `true`, если модуль входит в цепочку поиска, иначе возвращается `false`:

```
 > require_relative 'few_modules'
Title#title
=> true
 > Page.include? Seo
=> true
 > Page.include? Title
=> true
 > Page.include? Kernel
=> true
 > Page.include? Enumerable
=> false
```

Чтобы не перебирать все известные модули, можно воспользоваться методом `included_modules`, который возвращает список всех модулей из цепочки поиска метода:

```
 > require_relative 'few_modules'
Title#title
=> true
 > Page.included_modules
=> [Title, Seo, Kernel]
```

20.8. Метод *prepend*

Методы класса перекрывают методы модуля, который включается в класс при помощи `include`. Эту ситуацию можно изменить, если заменить метод `include` на `prepend` (рис. 20.13).

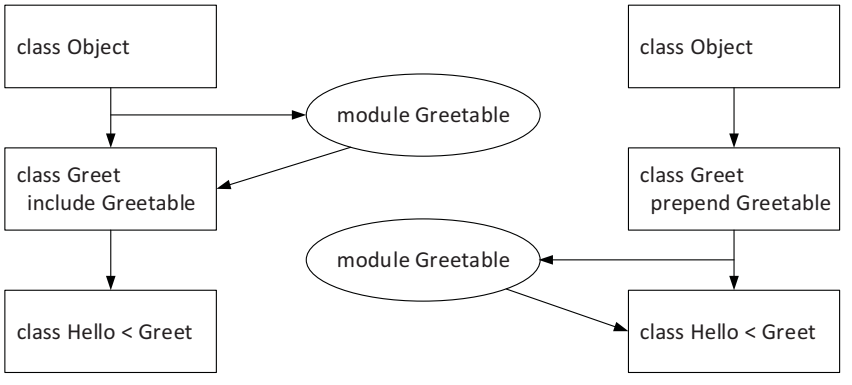


Рис. 20.13. Поиск метода при подключении модуля методами `include` и `prepend`

В листинге 20.32 модуль `Greetable` подключается в класс при помощи метода `prepend`.

Листинг 20.32. Подключение модуля при помощи `prepend`. Файл prepend.rb

```
class Greet
  def say(name)
    "Greet#say: Hello, #{name}!"
  end
end

module Greetable
  def say(name)
    "Greetable#say: Hello, #{name}!"
  end
end

class Hello < Greet
  prepend Greetable

  def say(name)
    "Hello#say: Hello, #{name}!"
  end
end

hello = Hello.new
puts hello.say('Ruby') # Greetable#say: Hello, Ruby!
p Hello.ancestors # [Greetable, Hello, Greet, Object, Kernel, BasicObject]
```

Как видно из приведенного примера, `prepend` разместил модуль `Greetable` первым в цепочке поиска методов. Если бы модуль `Greetable` подмешивался на уровень класса при помощи `include`, цепочка поиска метода выглядела бы так:

```
[Hello, Greetable, Greet, Object, Kernel, BasicObject]
```

Таким образом, используя `prepend` или `include`, можно разместить модуль в цепочке поиска либо до, либо после класса.

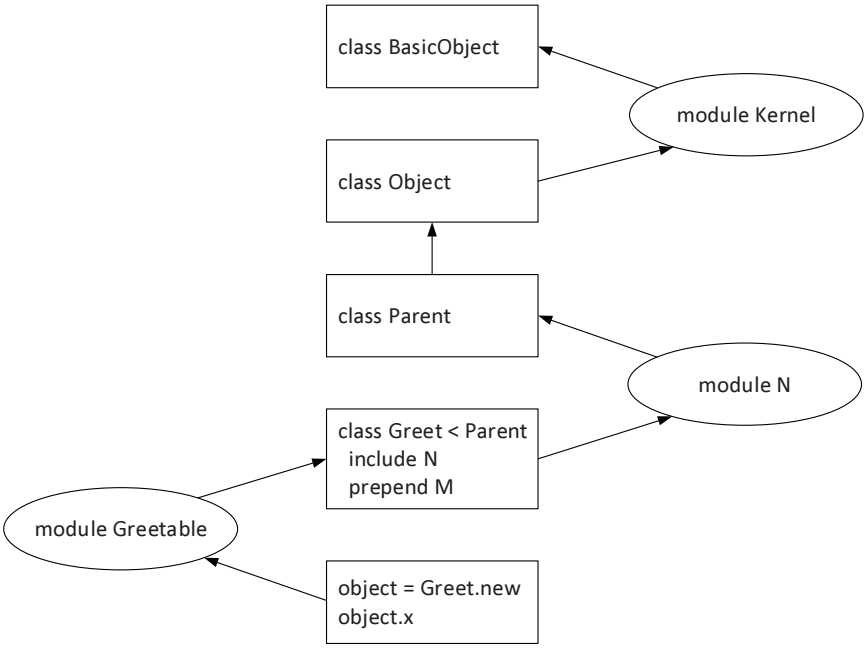


Рис. 20.14. Поиск метода в модулях и базовых классах

На рис. 20.14 осуществляется поиск метода `x`. При его поиске сначала опрашивается модуль `M`, подключенный при помощи `prepend`, затем класс, потом модуль `N`, подключенный при помощи `include`. Лишь после этого поиск осуществляется в базовом классе `C`. Если метод и в нем не будет обнаружен, поиск продолжится дальше: `Object`, `Kernel` и `BasicObject`.

20.9. Методы обратного вызова

Нам уже хорошо известен метод `initialize`, при помощи которого можно инициировать объект. Этот метод автоматически вызывается при создании объекта.

Метод `initialize` разработчик создает как обычный метод, однако сам его нигде не вызывает. Метод используется неявно во время создания объекта методом `new`. Такие методы называют *методами обратного вызова* или, более коротко, *хуками* (от англ. *hook* — якорь, крюк).

Метод `initialize` используется очень часто, но это не единственный метод обратного вызова, который предоставляет Ruby. Например, метод `at_exit` позволяет зарегистрировать обработчики для события завершения Ruby-программы (см. *разд. 9.8.2*).

Многие события в жизненном цикле модулей также могут быть снабжены обработчиками. В табл. 20.1 приводится список методов обратного вызова, которые обслуживают модули и классы.

***Таблица 20.1.** Методы обратного вызова для обслуживания модулей и классов*

Метод	Описание
`extended`	Вызывается при подмешивании модуля в класс при помощи метода `extend`
`included`	Вызывается при подмешивании модуля в класс при помощи метода `include`
`prepended`	Вызывается при подмешивании модуля в класс при помощи метода `prepend`
`method_added`	Вызывается при добавлении в модуль инстанс-метода
`method_removed`	Вызывается при удалении инстанс-метода из модуля — например, при помощи метода `remove_method`
`singleton_method_added`	Вызывается при добавлении в модуль синглетон-метода
`singleton_method_removed`	Вызывается при удалении синглетон-метода из модуля

Методы из табл. 20.1 реализуются в модуле в виде синглетон-методов и автоматически вызываются при возникновении события. Методы `extended`, `included` и `prepended` принимают параметр, через который передается класс, модуль или объект, в который включается модуль (листинг 20.33).

Листинг 20.33. Использование методов обратного вызова. Файл hook.rb

```
module Hook
  class << self
    def extended(mod)
      puts "Модуль включен в #{mod} при помощи extend"
    end

    def included(mod)
      puts "Модуль включен в #{mod} при помощи include"
    end

    def prepended(mod)
      puts "Модуль включен в #{mod} при помощи prepend"
    end
  end
end
```

```
class Ticket
  include Hook # Модуль включен в Ticket при помощи include
  extend Hook  # Модуль включен в Ticket при помощи extend
end

class Page
  prepend Hook # Модуль включен в Page при помощи prepend
  extend Hook  # Модуль включен в Page при помощи extend
end

ticket = Ticket.new
ticket.extend Hook
# Модуль включен в #<Ticket:0x00007f8e3b89cdc8> при помощи extend
```

Существует метод `extend_object`, который очень похож на `extended`. Этот метод можно перегрузить, и он тоже будет вызван при включении модуля при помощи `extend`. Однако в методе `extend_object` потребуется использовать ключевое слово `super`. Если этого не сделать, модуль не будет включен (листинг 20.34).

Листинг 20.34. Файл extend_object_prevent.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end

  def self.extend_object(obj)
    puts 'Предотвращение включения модуля'
  end
end

ticket = Object.new
ticket.extend Hello # Предотвращение включения модуля

puts ticket.say('Ruby') # undefined method `say'
```

Если в модуль `Hello` добавить метод обратного вызова `extended`, он тоже не будет вызван, т. к. расширение объекта `ticket` модулем `Hello` не произошло. Такой прием можно использовать для предотвращения включения модуля при помощи `extend`. Для штатной работы механизма включения модуля придется задействовать ключевое слово `super` (листинг 20.35).

Листинг 20.35. Штатное использование метода `extend_object`. Файл extend_object.rb

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
```

```
  class << self
    def extend_object(obj)
      puts 'Вызов Hello#extend_object'
      super
    end

    def extended(obj)
      puts 'Вызов Hello#extended'
      super
    end
  end
end

ticket = Object.new
ticket.extend Hello

puts ticket.say('Ruby') # Hello, Ruby!
```

Результатом выполнения программы будут следующие строки:

```
Вызов Hello#extend_object
Вызов Hello#extended
Hello, Ruby!
```

Таким образом, сначала выполняется метод `extend_object` и лишь после успешного расширения объекта методами модуля срабатывает метод обратного вызова `extended`.

На практике к `extend_object` прибегают лишь в том случае, если необходимо предотвратить успешное выполнение операции `extend`. Во всех остальных случаях рекомендуется использовать метод `exteneded`.

Методы `extended`, `included` и `prepended` часто используются в метапрограммировании, когда необходимо внести изменения в стандартные механизмы подключения модулей.

При включении модуля в класс при помощи `include`, среди одноименных методов класса и модуля выбирается метод класса (листинг 20.36).

Листинг 20.36. Будет использован метод `Ticket#say`. Файл prepend_wrong.rb

```
class Ticket
  def say(name)
    "Ticket: Hello, #{name}!"
  end
end

module Hello
  def say(name)
```

```
    "Hello: Hello, #{name}!"
  end
end

class Ticket
  include Hello
end

o = Ticket.new
puts o.say('world') # Ticket: Hello, world!
```

Ситуацию можно исправить, используя `prepend`:

```
class Ticket
  prepend Hello
end
```

Однако до версии 2.0 в Ruby не было метода `prepend`. Из ситуации можно было выкрутиться при помощи метода `included` (листинг 20.37).

Листинг 20.37. Эмуляция `prepend`. Файл prepend_emulate.rb

```
class Ticket
  def say(name)
    "Ticket: Hello, #{name}!"
  end
end

module Hello
  def self.included(cls)
    cls.class_eval do
      def say(name)
        "Hello: Hello, #{name}!"
      end
    end
  end
end

class Ticket
  include Hello
end

o = Ticket.new
puts o.say('world') # Hello: Hello, world!
```

Здесь в методе `included` к объекту класса `Ticket` применяется метод `class_eval`, позволяющий выполнить Ruby-код в контексте класса. Это позволяет переопределить уже существующий метод `say`.

Метод `method_added` позволяет реагировать на добавление метода в модуль (листинг 20.38).

Листинг 20.38. Использование `method_added`. Файл method_added.rb

```
module MethodTracker
  @@methods = []

  def say(name)
    "Hello, #{name}!"
  end

  def self.method_added(method)
    @@methods << method
    puts "Добавлен метод #{method}"
  end

  def version
    RUBY_VERSION
  end

  def list
    @@methods
  end

  def self.title
    'MethodTracker'
  end
end
```

Результатом выполнения программы будут следующие строки:

```
Добавлен метод version
Добавлен метод list
```

Метод `method_added` вызывается только для методов, которые добавляются в модуль после него. Кроме того, он вызывается лишь для инстанс-методов. Для синглетон-методов придется добавлять `method_added` в метакласс (листинг 20.39).

Листинг 20.39. Файл singleton_method_added.rb

```
module MethodTracker
  @@methods = []

  def say(name)
    "Hello, #{name}!"
  end
```

```
  def self.singleton_method_added(method)
    puts "Добавлен метод #{method}"
  end

  def version
    RUBY_VERSION
  end

  def list
    @@methods
  end

  def self.title
    'MethodTracker'
  end
end
```

Метод `singleton_method_added` является синглетон-методом и реагирует в том числе на собственное добавление в модуль. Результатом выполнения программы будут следующие строки:

```
Добавлен метод singleton_method_added
Добавлен метод title
```

20.10. Уточнения

В *разд. 13.5* использовалась техника открытия класса, при помощи которой можно модифицировать любой существующий класс: добавить, удалить или переопределить методы. В листинге 20.40 приводится пример открытия класса `String`, в который добавляется метод `hello`.

Листинг 20.40. Открытие класса `String`. Файл string_open.rb

```
class String
  def hello
    "Hello, #{self}!"
  end
end

puts 'Igor'.hello # Hello, Igor!
```

Теперь каждая строка в программе снабжена дополнительным методом `hello`. Внесенные в класс `String` изменения затрагивают все строки во всех классах, модулях и библиотеках.

Добавление нового метода — безобидная операция, однако удаление или переопределение методов может ломать стандартные классы и зависимые от них библиотеки.

Для исправления ситуации Ruby предоставляет механизм *уточнений*, который позволяет локализовать изменения внутри модуля, класса или файла.

Для реализации уточнений используется метод `refine`. Он размещается в отдельном модуле и принимает блок, в котором можно определить методы.

В листинге 20.41 приводится модуль `StringExt`, в котором при помощи `refine` расширяется стандартный класс `String`.

Листинг 20.41. Использование ключевого слова `refine`. Файл string_ext.rb

```
module StringExt
  refine String do
    def hello
      "Hello, #{self}!"
    end
  end
end

class Hello
  using StringExt

  def initialize(name)
    @name = name
  end

  def say
    @name.hello
  end
end

hello = Hello.new 'Igor'

puts hello.say    # Hello, Igor!
puts 'Igor'.hello # undefined method `hello' for "Igor":String
```

Модуль с `refine`-блоком подключается в класс при помощи метода `using`. В результате метод `hello` действует только на строки внутри класса `Hello`, не затрагивая строки за его пределами.

Действие `refine`-уточнений начинается после вызова `using` (рис. 20.15).

Допускается использование `refine` и в глобальной области видимости (листинг 20.42).

Листинг 20.42. `refine` в глобальной области видимости. Файл refine_global.rb

```
module StringExt
  refine String do
```

```
    def hello
      "Hello, #{self}!"
    end
  end
end

using StringExt

puts 'Igor'.hello # Hello, Igor!
```

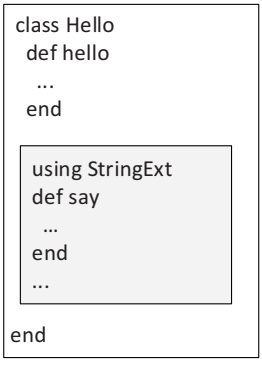


Рис. 20.15. Действие `using` начинается с точки вызова и продолжается до конца класса

Действие метода `using` распространяется только на текущий файл. Если попробовать подключить файл `refine_global` при помощи методов `require` или `require_relative`, действие `using` не будет распространяться на файл подключения (листинг 20.43).

Листинг 20.43. Файл refine_global_use.rb

```
require_relative 'refine_global'
puts 'Igor'.hello # undefined method `hello' for "Igor":String
```

Если потребуется, чтобы строки в новом файле тоже отзывались на метод `hello`, придется снова воспользоваться методом `using`.

20.11. Псевдонимы методов

Метод `alias_method` позволяет создать псевдонимы для методов. В качестве аргументов метод принимает символ с псевдонимом и символ с названием существующего метода. В листинге 20.44 приводится пример класса `Settings`, в котором при помощи метода `alias_method` создается псевдоним для метода `puts`. Затем метод `puts` переопределяется, однако, используя псевдоним `old_puts`, мы по-прежнему можем получить доступ к старой версии метода.

Листинг 20.44. Использование `alias_method`. Файл alias_method.rb

```
class Settings
  def initialize
    @list = {}
  end

  alias_method :old_puts, :puts

  def puts(*params)
    @list[params.first] = params[1]
  end

  def gets(key)
    @list[key]
  end

  def report
    @list.each do |k, v|
      old_puts "#{k} => #{v}"
    end
  end
end

s = Settings.new

s.puts :title, 'Новости'
s.puts :per_page, 30

s.report
```

Класс `Settings` предназначен для хранения настроек вида «ключ-значение». Для этого в нем заводится инстанс-переменная `@list`, которая хранит хэш с настройками, а также заводятся еще несколько методов: метод `puts` позволяет добавить новый параметр, метод `gets` — извлечь, а метод `report` — вывести список всех существующих параметров.

Результатом выполнения программы будут следующие строки:

```
title => Новости
per_page => 30
```

В методе `report` для вывода элемента используется псевдоним `old_puts`. Если вместо него попробовать использовать оригинальный метод `puts`, программа завершится ошибкой:

```
can't add a new key into hash during iteration (RuntimeError)
```

Дело в том, что метод `puts` переопределен в классе и экранирует одноименный метод для вывода в стандартный поток.

Задания

1. Вместо атрибутов `meta_title`, `meta_description` и `meta_keywords` создайте класс `Seo` с атрибутами `title`, `description` и `keywords`. В классы `News` и `About` добавьте SEO-информацию с использованием класса `Seo` (см. рис. 20.5).

2. Пусть имеется хэш `COLORS` с цветами радуги: ключом выступает название цвета на английском языке, значением — на русском. В глобальной области создайте методы `red`, `orange`, ..., `violete`, которые бы возвращали соответствующее название цвета на русском языке:

```
COLORS = {
  red: 'красный',
  orange: 'оранжевый',
  yellow: 'желтый',
  green: 'зеленый',
  blue: 'голубой',
  indigo: 'синий',
  violet: 'фиолетовый'
}
```

3. Реализуйте модуль `Fivable`, включение которого в класс будет разрешать создание только пяти объектов этого класса.

4. Создайте модуль `RomanNumbers` с методом `roman`, который переводит арабские цифры в римские. Расширьте класс `Integer` созданным модулем.

ГЛАВА 21

Стандартные модули

Файлы с исходными кодами этой главы находятся в каталоге *modules* сопровождающего книгу электронного архива.

Язык Ruby предоставляет ряд готовых модулей. С одним из них — `Kernel` — мы познакомились в *главе 20*. В этой главе будут рассмотрены другие популярные модули.

21.1. Модуль *Math*

Модуль `Math` предоставляет ряд констант, полезных в математических вычислениях. В листинге 21.1 при помощи констант `Math::PI` и `Math::E` выводятся числа π и e.

Листинг 21.1. Математические константы π и e. Файл math_consts.rb

```
puts Math::PI # 3.141592653589793
puts Math::E  # 2.718281828459045
```

Как можно видеть, нет необходимости заводить свои собственные константы для известных математических величин.

Метод `Math.sqrt` позволяет получить квадратный корень. В листинге 21.2 вычисляется расстояние между двумя точками (3, 7) и (–1, 5) в двумерной системе координат.

Листинг 21.2. Использование метода `Math.sqrt`. Файл distance.rb

```
puts Math.sqrt((3 + 1) ** 2 + (7 - 5) ** 2) # 4.47213595499958
```

Для вычисления квадратного корня можно применить оператор возведения в степень `**`, используя в качестве правого операнда дробное значение `0.5`:

```
 > Math.sqrt 2
=> 1.4142135623730951
```

```
 > 2 ** 0.5
=> 1.4142135623730951
```

Метод `Math.exp` возвращает экспоненциальную степень e^x. Добиться такого же результата можно путем возведения константы `Math::E` в степень `x` (листинг 21.3).

Листинг 21.3. Использование метода `Math.exp`. Файл exp.rb

```
puts Math.exp(2)   # 7.38905609893065
puts Math::E ** 2  # 7.3890560989306495
```

Метод `Math.log` возвращает натуральный логарифм. Модуль `Math` предоставляет отдельный метод для вычисления десятичного логарифма `Math.log10` (листинг 21.4).

Листинг 21.4. Вычисление логарифмов. Файл log.rb

```
puts Math.log(15)   # 2.70805020110221
puts Math.log10(15) # 1.1760912590556813
```

Модуль `Math` предоставляет методы для тригонометрических вычислений. В них большое значение играет число π, которое равно половине длины окружности с единичным радиусом (рис. 21.1).

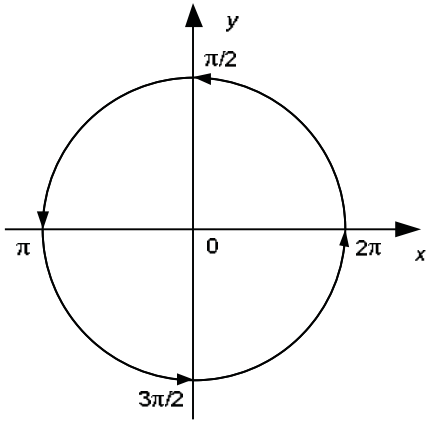


Рис. 21.1. Число π — это половина длины окружности с радиусом 1

Метод `Math.sin` возвращает синус, а метод `Math.cos` — косинус. Доступен так же метод для вычисления тангенса `Math.tan`. В качестве единственного аргумента методов передается угол, заданный в радианах (листинг 21.5).

Замечание

В модуле `Math` весьма много математических методов. Мы рассмотрели лишь их небольшую часть, с остальными придется познакомиться по документации.

Листинг 21.5. Тригонометрические вычисления. Файл trigonometry.rb

```
puts Math.sin(Math::PI / 2) # 1.0
puts Math.cos(Math::PI)     # -1.0
puts Math.tan(Math::PI / 4) # 0.9999999999999999
```

Обратите внимание, что метод `Math.tan` вместо единицы вернул значение `0.9999999999999999`. Это следствие бинарной природы наших компьютеров, в которых числа с плавающей точкой можно представить лишь приблизительно. В результате накапливается ошибка вычисления.

Схемы вычисления значения синуса и косинуса представлены на рис. 21.2 и 21.3 соответственно.

В листинге 21.5 на каждой строке упоминается модуль `Math`. Если в программе часто требуется обращаться к модулю, его можно подмешать в объект глобального

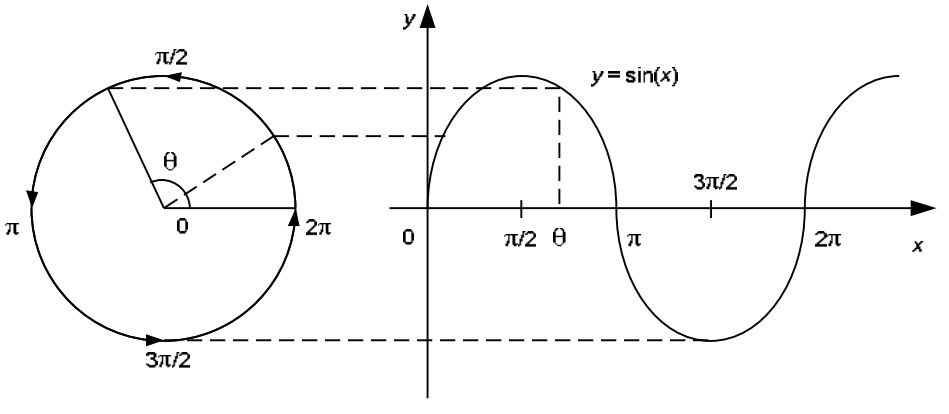


Рис. 21.2. Функция синуса *y* = sin(*x*). В качестве оси абсцисс *x* выступает угол θ, выраженный в радианах, т. е. в доле числа π, ордината *y* изменяется от –1 до 1

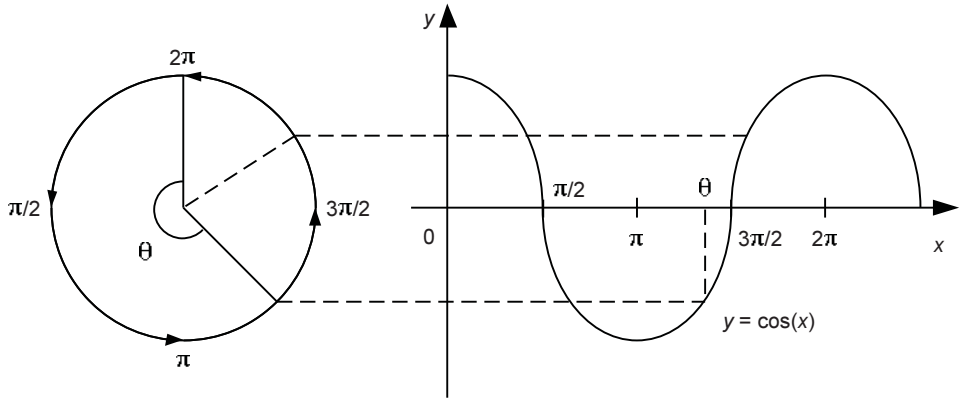


Рис. 21.3. Функция синуса *y* = cos(*x*). В качестве оси абсцисс *x* выступает угол θ, выраженный в радианах, т. е. в доле числа π, ордината *y* изменяется от –1 до 1

уровня при помощи метода include. Это позволит избавиться от префиксов Math. и Math:: (листинг 21.6).

Листинг 21.6. Подмешивание модуля Math. Файл math_include.rb

```
include Math

puts sin(PI / 2) # 1.0
puts cos(PI)     # -1.0
puts tan(PI / 4) # 0.9999999999999999
```

Для подмешивания методов модуля Math был выбран include, а не extend, т. к. необходимо получить доступ к синглетон-методам модуля. Если бы модуль содержал инстанс-методы, можно было воспользоваться extend.

21.2. Модуль *Singleton*

В *разд. 18.6* рассматривалась реализация паттерна проектирования «Одиночка» (Singleton). В паттерне для класса допускается существование объекта лишь в единственном экземпляре.

При создании класса метод new объявлялся закрытым, а методы clone и dup переопределялись таким образом, чтобы они всегда возвращали ссылку на существующий объект. Тем самым исключалась возможность создания объекта в обход метода instance, который следил за тем, чтобы объект был создан только один раз.

На практике для реализации паттерна «Одиночка» используют готовый модуль Singleton. Чтобы воспользоваться этим модулем, потребуется подключить его библиотеку при помощи метода require (листинг 21.7).

Листинг 21.7. Использование модуля Singleton. Файл settings.rb

```
require 'singleton'

class Settings
  include Singleton

  def initialize
    @list = {}
  end

  def [](key)
    @list[key]
  end

  def []=(key, value)
    @list[key] = value
  end
end
```

Класс в `Settings` предназначен для хранения разнообразных параметров. Для этого инстанс-переменную `@list` инициализируем хэшем. Кроме того, переопределяем квадратные скобки, адресовав методы `[]` и `[]=` инстанс-переменной `@list`. Это позволит обращаться с объектом как с хэшем.

Доступ к методу `Settings::new` закрыт, единственный способ создать объект класса — обратиться к методу `instance`:

```
 > require_relative 'settings'
=> true
 > s = Settings.new
NoMethodError (private method `new' called for Settings:Class)
 > s = Settings.instance
=> #<Settings:0x00007f9a750b6488 @list={}>
```

Метод `instance` создает объект при первом обращении, а при последующих попытках обратиться к нему он возвращает ссылку на уже созданный объект:

```
 > require_relative 'settings'
=> true
 > first = Settings.instance
=> #<Settings:0x00007f9a750b6488 @list={}>
 > second = Settings.instance
=> #<Settings:0x00007f9a750b6488 @list={}>
 > first == second
=> true
```

Теперь можно воспользоваться классом `Settings` для хранения настроек (листинг 21.8).

Листинг 21.8. Использование класса `Settings`. Файл settings_use.rb

```
require_relative 'settings'

setting = Settings.instance
setting[:title] = 'Новостной портал'
setting[:per_page] = 30

params = Settings.instance
p params[:title]    # "Новостной портал"
p params[:per_page] # 30
```

Можно не опасаться появления копии объекта — он всегда будет существовать в единственном экземпляре.

21.3. Модуль *Comparable*

Модуль `Comparable` предназначен для более удобной реализации операторов логического сравнения (см. табл. 7.4). Продемонстрируем проблему на примере класса билета `Ticket`. Пусть класс содержит единственную инстанс-переменную `@price` — цену билета (листинг 21.9).

Листинг 21.9. Класс билета `Ticket`. Файл ticket.rb

```
class Ticket
  attr_reader :price

  def initialize(price:)
    @price = price
  end
end
```

Если мы попробуем сравнить два билета стоимостью `500` и `600`, то потерпим неудачу — объекты билетов не поддерживают операции сравнения:

```
 > require_relative 'ticket'
=> true
 > first = Ticket.new(price: 500)
=> #<Ticket:0x00007f94312271b8 @price=500>
 > second = Ticket.new(price: 600)
=> #<Ticket:0x00007f9434006408 @price=600>
 > first > second
NoMethodError (undefined method `>')
 > first < second
NoMethodError (undefined method `<')
```

Ruby-интерпретатор «не знает», как мы хотим сравнивать билеты: по стоимости, по `object_id` или выберем какой-либо другой критерий. Сравнение пользовательских объектов необходимо запрограммировать самостоятельно. Переопределим методы `>` и `<` в классе `Ticket` (листинг 21.10).

Листинг 21.10. Реализация операторов > и <. Файл ticket_less_more.rb

```
class Ticket
  attr_reader :price

  def initialize(price:)
    @price = price
  end

  def >(ticket)
    price > ticket.price
  end
```

```
  def <(ticket)
    price < ticket.price
  end
end
```

Попробуем воспользоваться обновленным классом `Ticket`:

```
 > require_relative 'ticket_less_more'
=> true
 > first = Ticket.new(price: 500)
=> #<Ticket:0x00007f9431140038 @price=500>
 > second = Ticket.new(price: 600)
=> #<Ticket:0x00007f94312319b0 @price=600>
 > first > second
=> false
 > first < second
=> true
 > first >= second
NoMethodError (undefined method `>=')
 > first <= second
NoMethodError (undefined method `<=')
```

Как можно видеть, операторы сравнения `>` и `<` заработали, однако попытка воспользоваться операторами `>=` и `<=` снова терпит неудачу. Для полноценной поддержки операторов сравнения потребуется реализация почти всех операторов из табл. 7.4. При этом методы выглядят однообразно и занимают значительный объем класса.

Исправить ситуацию можно, включив в класс модуль `Comparable` и реализовав оператор `<=>` (листинг 21.11). Модуль `Comparable` доступен сразу — для его использования не требуется подключать в код программы дополнительные библиотеки.

Листинг 21.11. Использование модуля `Comparable`. Файл ticket_comparable.rb

```
class Ticket
  include Comparable
  attr_reader :price

  def initialize(price:)
    @price = price
  end

  def <=>(ticket)
    price <=> ticket.price
  end
end
```

Оператор `<=>` сравнивает значения друг с другом и возвращает три состояния:

- `-1` — первый операнд меньше второго;
- `0` — оба операнда равны;
- `1` — первый операнд больше второго.

Для чисел оператор уже реализован, поэтому мы можем использовать его в классе `Ticket` для сравнения цен:

```
 > require_relative 'ticket_comparable'
=> true
 > first = Ticket.new(price: 500)
=> #<Ticket:0x00007f9432024db0 @price=500>
 > second = Ticket.new(price: 600)
=> #<Ticket:0x00007f943117d2a8 @price=600>
 > first > second
=> false
 > first < second
=> true
 > first >= second
=> false
 > first <= second
=> true
 > first != second
=> true
```

Благодаря модулю `Comparable`, в отношении билета можно использовать любой оператор сравнения.

21.4. Модуль *Enumerable*

Модуль `Enumerable` позволяет снабдить класс методами коллекции. Так можно обучить свои собственные классы вести себя, как массивы. Для этого достаточно включить модуль `Enumerable` в класс и реализовать метод `each`.

После этого в отношении объектов класса можно использовать все остальные итераторы: `map`, `reduce`, `select`, `reject` и т. д.

В листинге 21.12 показан класс радуги `Rainbow`, представляющий коллекцию цветов. Класс включает стандартный модуль `Enumerable` и реализует метод `each`.

Листинг 21.12. Использование модуля `Enumerable`. Файл rainbow.rb

```
class Rainbow
  include Enumerable

  def each
    yield 'красный'
    yield 'оранжевый'
    yield 'желтый'
    yield 'зеленый'
    yield 'голубой'
    yield 'синий'
    yield 'фиолетовый'
  end
end
```

Самая простая реализация `each` заключается в вызове ключевого слова `yield` для каждого из цветов радуги. Метод семь раз будет уступать вычисление блоку, передавая каждый раз строку с цветом в качестве параметра (рис. 21.4).

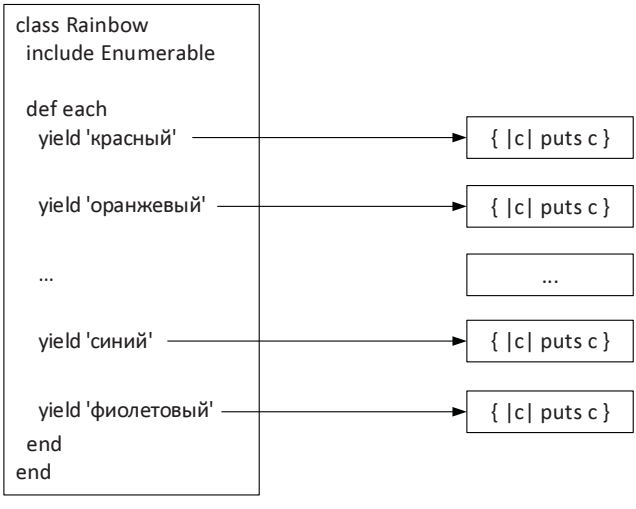


Рис. 21.4. Передача цветов через блок метода `Rainbow#each`

Класс `Rainbow` можно упростить, если заменить семь последовательных вызовов ключевого слова `yield` циклом или итератором (листинг 21.13).

Листинг 21.13. Переработка класса `Rainbow`. Файл rainbow_refactoring.rb

```
class Rainbow
  include Enumerable

  def initialize
    @colors = %w[красный оранжевый желтый зеленый
                 голубой синий фиолетовый]
  end

  def each
    @colors.each { |x| yield c }
  end
end
```

В листинге 21.14 создается объект класса `Rainbow` и выводится список цветов при помощи метода `each`.

Листинг 21.14. Вывод списка цветов при помощи метода `each`. Файл rainbow_each.rb

```
require_relative 'rainbow'

r = Rainbow.new
r.each { |c| puts c }
```

Результатом выполнения программы будет список цветов радуги:

```
красный
оранжевый
желтый
зеленый
голубой
синий
фиолетовый
```

Теперь можно воспользоваться любыми методами модуля `Enumerable`. Например, можно вывести массив названий цветов прописными (заглавными) буквами. Для этого удобно задействовать итератор `map`, применив к каждому из элементов коллекции метод `upcase` (листинг 21.15).

Листинг 21.15. Использование итератора `map`. Файл rainbow_map.rb

```
require_relative 'rainbow'

r = Rainbow.new
colors = r.map(&:upcase)
p colors
```

Результатом выполнения программы будет массив `colors` с названиями цветов в верхнем регистре:

```
["КРАСНЫЙ", "ОРАНЖЕВЫЙ", "ЖЕЛТЫЙ", "ЗЕЛЕНЫЙ", "ГОЛУБОЙ", "СИНИЙ", "ФИОЛЕТОВЫЙ"]
```

Модуль `Enumerable` содержит большое количество самых разнообразных методов. Например, при помощи метода `find` можно извлечь из коллекции первый элемент, для которого условие в блоке вернет истинное значение (листинг 21.16).

Листинг 21.16. Использование метода `find`. Файл rainbow_find.rb

```
require_relative 'rainbow'

r = Rainbow.new
puts r.find { |c| c.start_with? 'ж' } # желтый
```

В блоке метода `find` для каждого из параметров применяется логический метод `start_wtih?` , который проверяет, начинается ли строка с буквы `'ж'`. В результате метод `find` находит желтый цвет.

21.5. Модуль *Forwardable*

Модуль `Forwardable` позволяет делегировать методы объектам. В листинге 21.11 в классе `Ticket` переопределяется оператор `<=>`. Внутри метода опять вызывается оператор `<=>` в отношении инстанс-переменной `@price`:

```
class Ticket
  ...
  def <=>(ticket)
    price <=> ticket.price
  end
end
```

Оператор `<=>` можно делегировать при помощи метода `def_delegator` модуля `Forwardable`. Чтобы воспользоваться модулем, необходимо подключить его реализацию при помощи метода `require`, а также подмешать его в класс при помощи метода `extend` (листинг 21.17). Здесь мы применили метод `extend` вместо `include`, т. к. наша задача — задействовать методы модуля на уровне самого класса, а не его объектов.

Листинг 21.17. Использование метода `Forwardable`. Файл def_delegator.rb

```
require 'forwardable'

class Ticket
  attr_reader :price
  include Comparable
  extend Forwardable
  def_delegator :@price, :<=>, :<=>

  def initialize(price:)
    @price = price
  end
end

first = Ticket.new(price: 500)
second = Ticket.new(price: 600)

if first > second.price
  puts 'Первый билет стоит дороже второго'
elsif first < second.price
  puts 'Первый билет стоит дешевле второго'
else
  puts 'Билеты стоят одинаково'
end
```

Метод `def_delegator` позволяет не только делегировать вызовы методов другому объекту, но и изменять названия методов. В листинге 21.18 приводится пример класса для хранения настроек `Settings`, в нем при помощи `def_delegator` вводятся псевдонимы `set` и `get` для заполнения и извлечения параметров. Обращения к методам `set` и `get` делегируются инстанс-переменной `@list`. Вместо `set` вызывается оператор `[]=`, вместо `get` — оператор `[]`.

Листинг 21.18. Файл settings_delegator.rb

```
require 'singleton'
require 'forwardable'

class Settings
  include Singleton
  extend Forwardable
  def_delegator :@list, :[]=, :set
  def_delegator :@list, :[], :get

  def initialize
    @list = {}
  end
end
setting = Settings.instance
setting.set(:title, 'Новостной портал')
setting.set(:per_page, 30)

params = Settings.instance
p params.get(:title)    # "Новостной портал"
p params.get(:per_page) # 30
```

Модуль `Forwardable` позволяет избавиться от явного переопределения методов.

Помимо `def_delegator`, модуль `Forwardable` предоставляет метод `def_delegators`, позволяющий назначить сразу несколько методов. Эту форму удобно использовать в том случае, если нет необходимости изменять названия методов.

Переработаем класс `Settings` таким образом, чтобы он снова поддерживал методы `[]` и `[]=`, которые будут делегироваться на инстанс-переменной `@list` (листинг 21.19).

Листинг 21.19. Использование класса `Settings`. Файл settings_delegators.rb

```
require 'singleton'
require 'forwardable'

class Settings
  include Singleton
  extend Forwardable
  def_delegators :@list, :[]=, :[]

  def initialize
    @list = {}
  end
end
```

```
setting = Settings.instance
setting[:title] = 'Новостной портал'
setting[:per_page] = 30

params = Settings.instance
p params[:title]    # "Новостной портал"
p params[:per_page] # 30
```

Метод принимает в качестве первого аргумента получатель, все последующие аргументы являются именами методов, которые необходимо делегировать получателю.

Метод `delegate` так же предназначен для массового делегирования. В качестве аргумента метод принимает хэш, в качестве ключей которого выступают имена методов, а в качестве значений — получатель (листинг 21.20).

Листинг 21.20. Использование метода `delegate`. Файл settings_delegate.rb

```
require 'singleton'
require 'forwardable'

class Settings
  include Singleton
  extend Forwardable
  delegate :[]= => :@list, :[] => :@list

  def initialize
    @list = {}
  end
end

setting = Settings.instance
setting[:title] = 'Новостной портал'
setting[:per_page] = 30

params = Settings.instance
p params[:title]    # "Новостной портал"
p params[:per_page] # 30
```

Модуль `Forwardable` не обязательно использовать напрямую. Можно обернуть его своим собственным модулем. В листинге 21.21 приводится пример создания модуля `Decorator`, который объединяет два метода: `def_delegator` и `def_delegators` — в один метод `delegate`.

Листинг 21.21. Модуль `Decorator`. Файл decorator.rb

```
require 'singleton'
require 'forwardable'
```

```
module Decorator
  def self.included(cls)
    cls.extend Forwardable
    cls.extend ClassMethods
  end

  module ClassMethods
    def decorate(*methods, to:, as: nil)
      if as
        def_delegator to, methods.first, as
      else
        def_delegators to, *methods
      end
    end
  end
end

class Settings
  include Singleton
  include Decorator
  decorate :[]=, :[], to: :@list
  decorate :join, to: '@list.map {|k, v| "#{k} #{v} " }', as: :report

  def initialize
    @list = {}
  end
end

setting = Settings.instance
setting[:title] = 'Новостной портал'
setting[:per_page] = 30

params = Settings.instance
p params[:title]    # "Новостной портал"
p params[:per_page] # 30

puts params.report  # title Новостной портал per_page 30
```

При помощи параметра `as:` можно назначить псевдоним для метода, при помощи параметра `to:` — получатель. Обратите внимание, что в качестве получателя могут выступать не только названия методов и инстанс-переменных. В строках можно указывать Ruby-выражения.

За счет использования в модуле `Decorator` метода обратного вызова `included` можно добиться, что модуль будет подключаться в класс при помощи метода `include` вместо метода `extend` (в случае оригинального модуля `Forwardable`).

21.6. Маршаллизация

Маршаллизация — это процесс сохранения состояния объекта в строковое представление для долговременного хранения. В процессе демаршаллизации можно восстановить объект из строкового представления. Следует учитывать, что маршаллизация не может сохранить и восстановить потоки ввода/вывода.

> ***Замечание***
>
> В других языках программирования вместо термина *маршаллизация* чаще применяется термин *сериализация*.

Маршаллизировать метод можно при помощи метода `dump` модуля `Marshal` (листинг 21.22).

Листинг 21.22. Использование метода `Marshal.dump`. Файл marshal_dump.rb

```
require_relative 'ticket'

ticket = Ticket.new price: 600
p Marshal.dump(ticket) # "\x04\bo:\vTicket\x06:\v@pricei\x02X\x02"
```

Для восстановления объекта можно использовать метод `Marshal.load` (листинг 21.23).

Листинг 21.23. Использование метода `Marshal.load`. Файл marshal_load.rb

```
require_relative 'ticket'

ticket = Ticket.new price: 600
p ticket      # #<Ticket:0x00007fa2c40b93c0 @price=600>
str = Marshal.dump(ticket)
t = Marshal.load(str)

p t           # #<Ticket:0x00007fa2c40b9140 @price=600>
p t == ticket # false
puts t.price  # 600
```

Объекты `ticket` и `t` — это разные объекты, расположенные в разных участках оперативной памяти. Однако состояние их инстанс-переменных одинаковое.

В том случае, когда необходимо вмешаться в процесс маршаллизации, в классе реализуют два метода обратного вызова:

- `marshal_dump` — метод, который вызывается в методе `dump` и формирует массив сохраняемых значений;
- `marshal_load` — метод, который вызывается в методе `load` и инициализирует объект.

Создадим класс `Person`, который содержит четыре инстанс-переменные:

- `@first_name` — имя;
- `@last_name` — фамилия;
- `@patronymic` — отчество;
- `@password` — пароль.

При маршаллизации будем сохранять лишь фамилию, имя и отчество. Пароль в строковое представление объекта попадать не будет. При восстановлении инстанс-переменной `@password` будет назначаться неопределенное значение `nil`.

Чтобы управлять процессом маршаллизации и демаршаллизации, в класс `Person` необходимо добавить методы `marshal_dump` и `marshal_load` (листинг 21.24).

Листинг 21.24. Управление маршаллизацией. Файл person.rb

```
class Person
  attr_accessor :first_name, :last_name, :patronymic
  attr_reader :password

  def initialize(first_name:, last_name:, patronymic:, password:)
    @first_name = first_name
    @last_name = last_name
    @patronymic = patronymic
    @password = password
  end

  def marshal_dump
    [@first_name, @last_name, @patronymic]
  end

  def marshal_load(list)
    @first_name, @last_name, @patronymic = list
  end
end

person = Person.new(
  first_name: 'Иван',
  last_name: 'Петрович',
  patronymic: 'Сидоров',
  password: 'qwerty'
)

str = Marshal.dump(person)
prsn = Marshal.load(str)
```

```
p prsn.first_name # "Иван"
p prsn.last_name  # "Петрович"
p prsn.patronymic # "Сидоров"
p prsn.password   # nil
```

Метод marshal_dump формирует массив параметров, которые необходимо подвергнуть маршаллизации. Метод marshal_load принимает в качестве параметра точно такой же массив. Используя данные из массива, метод инициализирует уже подготовленную копию объекта.

В том случае, если вы хотите самостоятельно создать объект при восстановлении, следует реализовать методы обратного вызова: _dump и _load. Метод _dump должен вернуть строковое представление объекта. В методе _load необходимо создать новый объект при помощи метода new (листинг 21.25). Метод _dump должен быть инстанс-методом, а _load — синглетон-методом класса.

Листинг 21.25. Использование методов _dump и _load. Файл person_dump_load.rb

```
class Person
  attr_accessor :first_name, :last_name, :patronymic
  attr_reader :password
  SEPARATOR = '#'

  def initialize(first_name:, last_name:, patronymic:, password:)
    @first_name = first_name
    @last_name = last_name
    @patronymic = patronymic
    @password = password
  end

  def _dump(_version)
    [@first_name, @last_name, @patronymic].join SEPARATOR
  end

  def self._load(list)
    first_name, last_name, patronymic = list.split SEPARATOR
    new(
      first_name: first_name,
      last_name: last_name,
      patronymic: patronymic,
      password: nil
    )
  end
end

person = Person.new(
  first_name: 'Иван',
  last_name: 'Петрович',
```

```
  patronymic: 'Сидоров',
  password: 'qwerty'
)

str = Marshal.dump(person)
prsn = Marshal.load(str)

p prsn.first_name # "Иван"
p prsn.last_name  # "Петрович"
p prsn.patronymic # "Сидоров"
p prsn.password   # nil
```

Маршаллизация была весьма популярным средством для долговременного хранения состояния в 90-х годах прошлого столетия. В настоящий момент маршаллизации и сериализации стараются избегать.

Маршаллизация опасна при работе с данными из внешнего источника. Существует опасность выполнения произвольного кода.

Кроме того, маршаллизованные объекты трудно использовать для обмена в кросс-языковой среде. С одной стороны, данные обрабатываются программами на разных языках программирования и разобрать маршаллизованный объект в другой языковой среде трудно. С другой стороны, двоичный формат, который генерирует модуль `Marshal`, зависит от версии Ruby и может изменяться в новых релизах.

21.7. JSON-формат

Вместо маршаллизации для хранения и обмена объектами чаще используются кросс-языковые форматы — например, JSON. Аббревиатура JSON раскрывается как JavaScript Object Notation — готовые JavaScript-объекты.

JSON — это текстовый формат, который хранит данные в виде коллекции элементов «ключ-значение». Ключи всегда являются строками, а в качестве значений могут выступать числа, строки, массивы и хэши. Более сложные данные строятся на основе этих значений. Например, дата и время почти всегда кодируются строкой. В листинге 21.26 приводится пример JSON-коллекции.

> ***Замечание***
>
> Помимо JavaScript, формат JSON используется для представления документов NoSQL базы данных MongoDB. Современные реляционные базы данных PostgreSQL и MySQL поддерживают JSON-поля. В связи с этим JSON приобрел широкую популярность для обмена данными, особенно в веб-программировании.

Листинг 21.26. Пример JSON-формата. Файл person.json

```
{
  "first_name": "Иван",
  "last_name": "Петрович",
  "patronymic": "Сидоров",
```

```
  "params": {
    "score": 20,
    "number": 762
  }
}
```

JSON-формат очень похож на `Hash`. Для того чтобы преобразовать его в хэш, можно воспользоваться методом `parse` модуля `JSON`. Для того чтобы воспользоваться модулем, необходимо его подключить при помощи метода `require` (листинг 21.27).

Листинг 21.27. Разбор JSON-формата. Файл json_parse.rb

```
require 'json'

params = JSON.parse('{"hello": "world", "language": "Ruby"}')

puts params['hello']    # world
puts params['language'] # Ruby
```

В том случае, если необходимо преобразовать в хэш содержимое JSON-файла, можно прочитать его содержимое методом `File.read`. В результате метод вернет содержимое файла в виде строки, которую можно передать методу `JSON.parse` (листинг 21.28).

Замечание

Работа с файлами и каталогами подробно описывается в *главах 27–29*.

Листинг 21.28. Разбор JSON-файла. Файл json_file_parse.rb

```
require 'json'

params = JSON.parse File.read('person.json')
p params
```

Результатом выполнения программы будет следующая строка:

```
{"first_name"=>"Иван", "last_name"=>"Петрович", "patronymic"=>"Сидоров",
                                        "params"=>{"score"=>20, "number"=>762}}
```

Хэш можно превратить в JSON-строку при помощи метода `to_json` (листинг 21.29).

Листинг 21.29. Преобразования хэша в JSON-строку. Файл hash_to_json.rb

```
require 'json'

person = {
  first_name: 'Иван',
  last_name: 'Петрович',
  patronymic: 'Сидоров',
```

```
  params: {
    score: 20,
    number: 762
  }
}

puts person.to_json
```

Результатом выполнения программы будет следующая JSON-строка:

```
{"first_name":"Иван","last_name":"Петрович","patronymic":"Сидоров",
                                              "params":{"score":20,"number":762}}
```

Метод появляется в Ruby-объектах только после подключения JSON-библиотеки методом `require`. Если этого не сделать, Ruby-интерпретатор выдаст сообщение об ошибке: не найден метод `to_json`.

Вместо медленной Ruby-реализации `json` можно подключить более быструю реализацию на языке Си — `json/ext` (листинг 21.30).

Листинг 21.30. Использование `json/ext`. Файл hash_to_json_ext.rb

```
require 'json/ext'
...
```

21.8. YAML-формат

YAML — еще один распространенный формат для хранения структурированных коллекций «ключ-значение». Акроним YAML означает «AML Ain't Markup Language» (YAML — не язык разметки). В листинге 21.31 приводится типичный YAML-файл.

Листинг 21.31. Пример YAML-формата. Файл person.yml

```
params:
  first_name: Иван
  last_name: Сидоров
  patronymic: Петрович
  params:
    score: 20
    number: 762
```

Для работы с YAML-форматом предназначен модуль `YAML`. Модуль предоставляет метод `dump` для создания YAML-файла из Ruby-объекта и метод `load` для его восстановления.

В отличие от JSON, модуль YAML позволяет сериализовать не только хэши, но и любые Ruby-объекты. Для демонстрации возможностей модуля создадим класс `Person` (листинг 21.32).

Листинг 21.32. Пример YAML-формата. Файл yaml_person.rb

```
class Person
  attr_accessor :first_name, :last_name, :patronymic, :password

  def initialize(first_name:, last_name:, patronymic:, password:)
    @first_name = first_name
    @last_name = last_name
    @patronymic = patronymic
    @password = password
  end
end
```

В листинге 21.33 создадим объект класса `Person` и сохраним его в строковое представление при помощи метода `YAML.dump`.

> ***Замечание***
>
> Модуль `YAML` — это псевдоним модуля `Psych`, для которого документация представлена более полно.

Листинг 21.33. Использование метода `YAML.dump`. Файл yaml_dump.rb

```
require 'yaml'
require_relative 'yaml_person'

person = Person.new(
  first_name: 'Иван',
  last_name: 'Петрович',
  patronymic: 'Сидоров',
  password: 'qwerty'
)

params = YAML.dump(person)

puts params
```

Результатом работы программы будут следующие строки:

```
--- !ruby/object:Person
first_name: Иван
last_name: Петрович
patronymic: Сидоров
password: qwerty
```

Из полученной строки в любой момент можно восстановить исходный объект — для этого предназначен метод `YAML.load` (листинг 21.34).

Листинг 21.34. Использование метода `YAML.load`. Файл yaml_load.rb

```
require 'yaml'
require_relative 'yaml_person'

yaml_str = <<~HERE
--- !ruby/object:Person
first_name: Иван
last_name: Петрович
patronymic: Сидоров
password: qwerty
HERE

person = YAML.load(yaml_str)

p person.first_name # "Иван"
p person.last_name  # "Петрович"
p person.patronymic # "Сидоров"
p person.password   # "qwerty"
```

Для работы с YAML-файлами предназначен метод `load_file`. В листинге 21.35 приводится программа, которая разбирает файл person.yml, формируя из него хэш.

Листинг 21.35. Использование метода `YAML.load_file`. Файл yaml_load_file.rb

```
require 'yaml'

person = YAML.load_file('person.yml')

puts person['params']['first_name']       # Иван
puts person['params']['last_name']        # Сидоров
puts person['params']['patronymic']       # Петрович
puts person['params']['params']['score']  # 20
puts person['params']['params']['number'] # 762
```

Модуль `YAML` предоставляет метод `to_json`, позволяющий получить JSON-представление объекта (листинг 21.36).

Листинг 21.36. Использование метода `YAML.to_json`. Файл yaml_json.rb

```
require 'yaml'
require_relative 'yaml_person'

person = Person.new(
  first_name: 'Иван',
  last_name: 'Петрович',
```

```
  patronymic: 'Сидоров',
  password: 'qwerty'
)

params = YAML.to_json(person)

puts params
```

Результатом работы программы будет следующее JSON-представление объекта:

```
{ first_name: "Иван", last_name: "Петрович", patronymic: "Сидоров",
                                                    password: "qwerty" }
```

Задания

1. Создайте класс `Group`, объект которого должен представлять коллекцию пользователей класса `User`. Задействуйте для реализации класса `Group` модуль `Enumerable`.
2. Создайте объект новостной заметки `News`, которая должна содержать название, тело новости и дату публикации. Задействуйте модуль `Comparable` и реализуйте сравнение новостей: чем более свежая новость, тем она «больше».
3. Откройте класс `Hash` и добавьте в него метод `to_json`, который возвращает строковое представление хэша в JSON-формате. Задачу следует решить, не прибегая к модулю `JSON`.

ГЛАВА 22

Свойства объектов

Файлы с исходными кодами этой главы находятся в каталоге *objects* сопровождающего книгу электронного архива.

Некоторые методы можно вызывать в отношении любого объекта. Даже самые простые объекты в Ruby обладают готовыми методами. Это связано с тем, что классы объектов наследуют свойства минимум двух классов: `BasicObject` и `Object`.

Большинство таких методов уже рассмотрены в предыдущих главах. Однако ряд важных механизмов: заморозка, пометка объекта — остались не раскрытыми. Именно им посвящена эта глава.

22.1. Общие методы

Любой класс, кроме `BasicObject`, имеет базовый класс, от которого наследуются свойства и методы (рис. 22.1).

Даже если класс не реализует ни одного метода, его объект содержит десятки готовых методов (листинг 22.1).

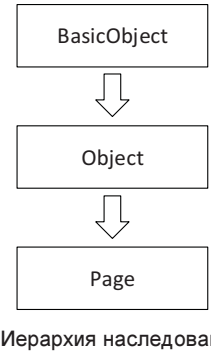


Рис. 22.1. Иерархия наследования классов

Листинг 22.1. Объект даже пустого класса реализует десятки методов. Файл ticket.rb

```
class Ticket
end

t = Ticket.new
puts t.methods.size # 58
```

Ряд таких методов нам уже знаком: `object_id`, `send`, `class`, `methods`, `nil?` и т. д.

22.2. Неизменяемые объекты

Некоторые объекты является неизменяемыми — например, символы и числа. Для них невозможно определить синглетон-методы (листинг 22.2).

Листинг 22.2. Строки и числа неизменяемы. Файл fixed.rb

```
number = 3
def number.say(name)
  "Hello, #{name}!"
end

symbol = :white
def symbol.say(name)
  "Hello, #{name}!"
end
```

Попытка определения метода на неизменяемых объектах завершается сообщениями об ошибке:

```
fixed.rb:2:in `<main>': can't define singleton (TypeError)
fixed.rb:7:in `<main>': can't define singleton (TypeError)
```

Большинство объектов может быть изменено (листинг 22.3).

Листинг 22.3. Изменение объекта массива. Файл changed.rb

```
arr = [1, 2, 3, 4, 5]
arr.delete_at(0)

p arr # [2, 3, 4, 5]

def arr.say(name)
  "Hello, #{name}!"
end

puts arr.say 'world' # Hello, world!
```

22.3. Заморозка объектов

Неизменяемым можно сделать любой объект языка Ruby — для этого достаточно воспользоваться методом `freeze` (листинг 22.4). Объекты, к которым применен метод `freeze`, называются *замороженными*.

Листинг 22.4. Использование метода `freeze`. Файл freeze.rb

```
arr = [1, 2, 3, 4, 5]

arr.freeze

arr.delete_at(0) # `delete_at': can't modify frozen Array (FrozenError)
p arr

def arr.say(name)  # can't modify frozen object (FrozenError)
  "Hello, #{name}!"
end
puts arr.say 'world'
```

Любая попытка изменить внутреннее состояние замороженного объекта завершается ошибкой.

Выяснить, является ли объект замороженным, можно при помощи логического метода `frozen?`, который возвращает `true`, если объект заморожен, и `false`, если объект можно изменять:

```
 > 1.frozen?
=> true
 > 'hello'.frozen?
=> false
 > o = Object.new
=> #<Object:0x007fce9488ac60>
 > o.frozen?
=> false
 > o.freeze
=> #<Object:0x007fce9488ac60>
 > o.frozen?
=> true
```

Замороженный объект нельзя вернуть в обычное состояние. Однако имеется возможность получить размороженную копию объекта — при клонировании его методом `dup` (листинг 22.5).

Листинг 22.5. Получение размороженной копии объекта. Файл dup.rb

```
first = Object.new
first.freeze
```

```
second = first.dup

p first.frozen?  # true
p second.frozen? # false
```

Разумеется, строки и символы остаются неизменными при попытке их клонирования. На самом деле в таком случае возвращается оригинальный объект:

```
 > 1.dup.frozen?
=> true
 > :white.dup.frozen?
=> true
```

При клонировании объекта методом `clone` копия также остается замороженной (листинг 22.6).

Листинг 22.6. Использование метода `clone`. Файл clone.rb

```
first = Object.new
first.freeze

second = first.clone

p first.frozen?  # true
p second.frozen? # true
```

22.4. Небезопасные объекты

Любой пользовательский ввод рассматривается как ненадежный. Пользователь может ошибочно вводить неправильные данные. Злоумышленники могут специально подбирать такой набор данных, который будет нарушать работу программы, выполнять произвольный код, похищать конфиденциальную информацию.

К любому вводу, который поступает извне, нужно относиться с удвоенным вниманием. Для облегчения этой задачи в Ruby любой объект допускается пометить как ненадежный. Объявление объекта ненадежным осуществляется путем вызова метода `taint`. Проверить надежность объекта можно при помощи логического метода `tainted?` (листинг 22.7).

> ***Замечание***
>
> Ruby-объекты поддерживают устаревшие методы `trust`, `untrust` и `untrusted?`. Вместо них следует использовать `untaint`, `taint` и `tainted?`.

Листинг 22.7. Объявление объекта ненадежным. Файл taint.rb

```
o = Object.new

p o.tainted? # false
o.taint
p o.tainted? # true
```

Копии объектов, полученные методами `clone` и `dup`, сохраняют отметку о ненадежности (листинг 22.8).

Листинг 22.8. Клоны ненадежного объекта тоже ненадежны. Файл taint_clone.rb

```
o = Object.new

o.taint

first = o.clone
second = o.dup

p o.tainted?      # true
p first.tainted?  # true
p second.tainted? # true
```

Объекты, полученные из ненадежного объекта, тоже рассматриваются как ненадежные (листинг 22.9).

Листинг 22.9. Файл taint_derivative.rb

```
hello = 'Hello, world!'

hello.taint

p hello.tainted?
p hello.upcase.tainted?
p hello[2..3].tainted?
```

Ruby-интерпретатор старается сразу пометить как ненадежные аргументы командной строки и переменные окружения (листинг 22.10).

Листинг 22.10. Файл auto_taint.rb

```
p ARGV.first.tainted?  # true
p ENV['PATH'].tainted? # true
```

Программу следует запустить хотя бы с одним дополнительным параметром:

```
$ ruby auto_taint.rb hello
```

В противном случае выражение `ARGV.first` вернет объект `nil`, который рассматривается как надежный, и логический метод `tainted?` вернет `false`.

Данные, полученные от пользователя методом `gets`, также рассматриваются как ненадежные (листинг 22.11).

Листинг 22.11. Файл taint_gets.rb

```
print 'Пожалуйста, введите произвольную строку '
value = gets
p value.tainted? # true
```

Объект можно объявить надежным при помощи метода `untaint`. В листинге 22.12 от пользователя принимается строка методом `gets`. Объект строки автоматически помечается как ненадежный. После того, как из строки удаляются все пробельные символы, она по-прежнему считается ненадежной. При помощи метода `untaint` можно пометить этот объект как надежный.

Листинг 22.12. Использование метода `untaint`. Файл untaint.rb

```
print 'Пожалуйста, введите произвольную строку '
value = gets

str = value.gsub(' ', '')
p str.tainted? # true
str.untaint
p str.tainted? # false
```

Задания

1. Создайте класс куба в трехмерном пространстве. Добейтесь, чтобы для кубов с одинаковыми координатами и размерами в программе всегда существовал единственный объект.
2. Запросите у пользователя программы десять строк, состоящих только из символов русского и английского алфавитов. Другие символы: цифры, пробелы, знаки препинания — не допускаются. Полученный список слов выведите в алфавитном порядке. Если одно и то же слово введено несколько раз, в списке выводим его один раз, однако рядом в скобках указываем, сколько раз слово было введено пользователем.

ГЛАВА 23

Массивы

Файлы с исходными кодами этой главы находятся в каталоге *arrays* сопровождающего книгу электронного архива.

Массив — это объект, представляющий коллекцию других объектов. Элементы коллекции проиндексированы и хранятся в заданном при ее создании порядке. Доступ к отдельным элементам массива можно получить по индексу, который начинается с нуля.

Мы уже затрагивали работу с массивами в *разд. 4.7*, *разд. 14.2.2* и *главе 11*. В этой главе мы изучим их более детально.

23.1. Модуль *Enumerable*

Массивы тесно связаны с модулем `Enumerable` (см. *разд. 21.4*). Этот модуль предоставляет большое количество инстанс-методов:

```
> Enumerable.instance_methods
=> [:to_h, :include?, :to_set, :max, :min, :find, :to_a, :entries, :sort,
:sort_by, :grep, :grep_v, :count, :detect, :find_index, :find_all, :select,
:filter, :reject, :collect, :map, :flat_map, :collect_concat, :inject, :reduce,
:partition, :group_by, :first, :all?, :any?, :one?, :none?, :minmax, :min_by,
:max_by, :minmax_by, :member?, :each_with_index, :reverse_each, :each_entry,
:each_slice, :each_cons, :each_with_object, :zip, :take, :take_while, :drop,
:drop_while, :cycle, :chunk, :slice_before, :slice_after, :slice_when,
:chunk_while, :sum, :uniq, :chain, :lazy]
```

Все эти методы автоматически становятся доступны объектам класса, в который включается модуль `Enumerable` и в котором реализован метод `each`. В листинге 23.1 приводится класс `Numerator` — в него подмешан модуль `Enumerable`, и в нем реализован метод `each`. Метод `each` при помощи ключевого слова `yield` пять раз уступает вычисления внешнему блоку, передавая в качестве параметров цифры от 1 до 5.

Листинг 23.1. Использование модуля `Enumerable`. Файл enumerable.rb

```
class Numerator
  include Enumerable
```

```
  def each
    yield 1
    yield 2
    yield 3
    yield 4
    yield 5
  end
end

n = Numerator.new

p n.select(&:even?) # [2, 4]
p n.max # 5
```

Как нам известно из *разд. 4.7*, массивы — это объекты класса `Array`:

```
 > [].class
=> Array
```

Если мы воспользуемся методом `ancestors`, то в цепочке наследования для класса `Array` можно обнаружить модуль `Enumerable`:

```
 > Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]
```

Это означает, что все методы модуля `Enumerable` доступны и для массивов (листинг 23.2).

Листинг 23.2. Использование модуля `Enumerable`. Файл enumerable_array.rb

```
arr = [1, 2, 3, 4, 5]

p arr.select(&:even?) # [2, 4]
p arr.max # 5
```

Поэтому при изучении документации массивов не следует обходить вниманием методы модуля `Enumerable`.

23.2. Заполнение массива

Массив можно заполнить при его создании. В листинге 23.3 приводятся различные способы создания заполненного массива.

Листинг 23.3. Создание массива. Файл array.rb

```
p [1, 2, 3, 4, 5]           # [1, 2, 3, 4, 5]
p [*1..5]                   # [1, 2, 3, 4, 5]
p (1..5).to_a               # [1, 2, 3, 4, 5]
p Array.new(5, 1)           # [1, 1, 1, 1, 1]
```

```
p Array.new(5) { |i| i + 1 } # [1, 2, 3, 4, 5]
p %w[1 2 3 4 5]              # ["1", "2", "3", "4", "5"]
p %i[1 2 3 4 5]              # [:"1", :"2", :"3", :"4", :"5"]
```

Заполнять массивы допускается и после их создания — для этого можно воспользоваться оператором присваивания. В листинге 23.4 в пустой массив `arr` добавляются первые три элемента (нумерация начинается с нуля).

Листинг 23.4. Заполнение массива. Файл assign.rb

```
arr = []
arr[0] = 1
arr[1] = 2
arr[2] = 3

p arr # [1, 2, 3]
```

При каждом новом присваивании новый элемент помещается в конец массива (рис. 23.1).

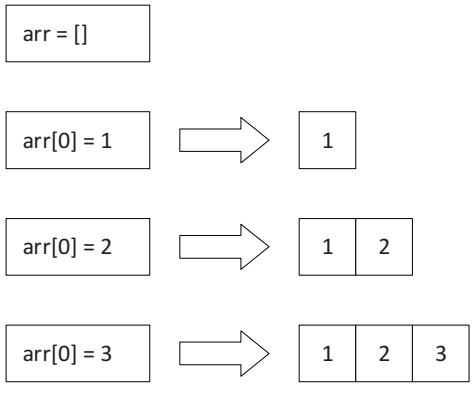


Рис. 23.1. Новые элементы добавляются в конец массива

Можно присвоить значение сразу 10-му элементу — т. е. элементу с индексом 9. При этом все элементы — от первого до 8-го — получат неопределенное значение `nil` (листинг 23.5).

Листинг 23.5. Заполнение массива. Файл sparse.rb

```
arr = []
arr[9] = 1

p arr # [nil, nil, nil, nil, nil, nil, nil, nil, nil, 1]
```

При обращении к несуществующему элементу массива Ruby-интерпретатор тоже возвращает значение `nil`:

```
 > arr = []
=> []
 > arr[100]
=> nil
```

В отличие от многих других языков программирования, в Ruby при добавлении нового элемента в массив нельзя использовать пустые квадратные скобки:

```
> arr[] = 1
ArgumentError (wrong number of arguments (given 1, expected 2))
```

Следует помнить, что оператор `[]=` — это на самом деле метод, который ожидает два аргумента. Если один из аргументов не передать — возникнет синтаксическая ошибка:

```
 > arr.[]=(0, 1)
=> 1
 > arr
=> [1]
```

Добавить элемент в конец массива можно при помощи оператора << (листинг 23.6).

Листинг 23.6. Использование оператора <<. Файл add.rb

```
arr = []
arr << 1
arr << 2
arr << 3

p arr # [1, 2, 3]
```

Помимо оператора <<, для заполнения массива можно использовать метод `push`. Его отличие от оператора << заключается в том, что он может принимать неограниченное количество аргументов (листинг 23.7).

> ***Замечание***
> Метод `push` имеет синоним `append`.

Листинг 23.7. Использование метода push. Файл push.rb

```
arr = []
arr.push 1
arr.push 2, 3, 4, 5 # [1, 2, 3, 4, 5]

p arr
```

Элементы можно добавлять не только в конец, но и в начало массива (рис. 23.2). Следует учитывать, что операция добавления элемента в начало массива — затратная по ресурсам и требует создания копии массива.

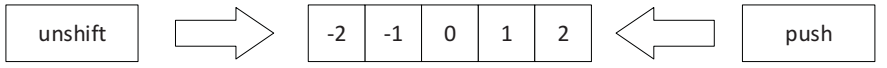


Рис. 23.2. Добавление элементов в начало и в конец массива

Добавить элемент в начало массива можно при помощи метода `unshift` (листинг 23.8).

Замечание

С версии Ruby 2.5 метод `unshift` имеет синоним `prepend`.

Листинг 23.8. Использование метода `unshift`. Файл unshift.rb

```
arr = [1, 2, 3]
arr.unshift(0)
arr.unshift(-3, -2, -1)

p arr # [-3, -2, -1, 0, 1, 2, 3]
```

Как и метод `push`, метод `unshift` может принимать произвольное количество аргументов.

Для вставки в произвольную точку массива предназначен метод `insert`. В качестве первого аргумента метод принимает индекс массива. Второй и последующие аргументы будут рассматриваться как элементы для вставки (листинг 23.9).

Листинг 23.9. Использование метода `insert`. Файл insert.rb

```
arr = %w[a b c]

arr.insert(0, 1)
p arr # [1, "a", "b", "c"]

arr.insert(2, 2, 3)
p arr # [1, "a", 2, 3, "b", "c"]

arr.insert(-1, 4)
p arr # [1, "a", 2, 3, "b", "c", 4]

arr.insert(-3, 5)
p arr # [1, "a", 2, 3, "b", 5, "c", 4]
```

Можно использовать отрицательные индексы — в этом случае они отсчитываются не от начала, а от конца массива (рис. 23.3).

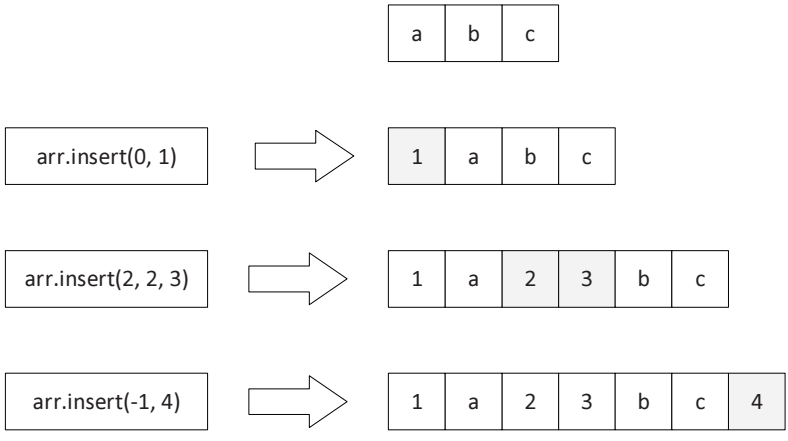


Рис. 23.3. Вставка значений в произвольную точку массива

23.3. Извлечение элементов

Извлекать элементы массива можно при помощи квадратных скобок, в которых указывается элемент массива (листинг 23.10). Если указывается отрицательное значение, отсчет индекса ведется с конца массива.

Листинг 23.10. Извлечение элементов массива. Файл bracket.rb

```
arr = [1, 2, 3, 4, 5]

puts arr[0]  # 1
puts arr[4]  # 5
puts arr[-1] # 5
puts arr[-2] # 4

p arr[1..3]  # [2, 3, 4]
p arr[1...3] # [2, 3]
p arr[1..-2] # [2, 3, 4]

p arr[1, 1]  # [2]
p arr[2, 1]  # [3]
p arr[2, 2]  # [3, 4]
p arr[-2, 2] # [4, 5]
```

Срезом называют часть элементов массива, которые следуют друг за другом без разрывов. Иногда вместо термина срез используется термин *подмассив*.

Если внутри квадратных скобок указывается диапазон, извлекается срез массива с индексами, входящими в диапазон. В диапазонах также допускается использование отрицательных индексов (рис. 23.4).

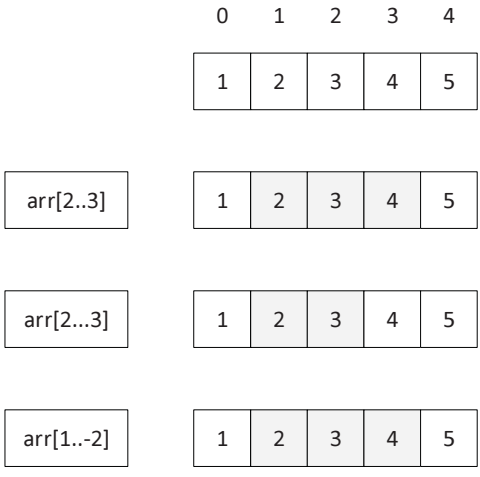


Рис. 23.4. Использование диапазонов для извлечения срезов массива

Квадратные скобки допускают передачу двух параметров через запятую: первый параметр является индексом, от которого ведется отсчет, второй параметр задает количество извлекаемых элементов (рис. 23.5).

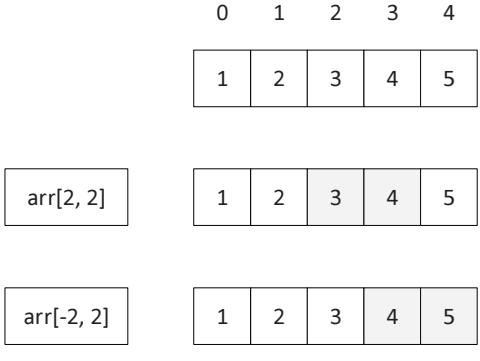


Рис. 23.5. Использование второго параметра для извлечения срезов массива

Операции, которые выполняются при помощи квадратных скобок, дублированы методами. Например, метод `at` принимает в качестве аргумента индекс элемента и возвращает его значение (листинг 23.11).

Замечание

Методы, в названии которых имеется префикс `at`, в качестве аргументов, как правило, принимают индексы.

Листинг 23.11. Использование метода `at`. Файл at.rb

```
arr = [1, 2, 3, 4, 5]

puts arr.at(2)  # 3
puts arr.at(-2) # 4
```

При обращении к несуществующему элементу метод `at` возвращает неопределенное значение `nil`.

Если необходимо, чтобы обращение к несуществующему массиву приводило к сообщению об ошибке, можно воспользоваться методом `fetch` (листинг 23.12).

Листинг 23.12. Использование метода `fetch`. Файл fetch.rb

```
arr = [1, 2, 3, 4, 5]

puts arr.fetch(2)  # 3
puts arr.fetch(-2) # 4
puts arr.fetch(10) # index 10 outside of array bounds: -5...5
```

Метод `fetch` допускает использование второго необязательного аргумента. В случае выхода за границы массива, вместо ошибки метод будет возвращать значение второго аргумента (листинг 23.13).

Листинг 23.13. Файл fetch_param.rb

```
arr = [1, 2, 3, 4, 5]

p arr.fetch(2, 0)    # 3
p arr.fetch(10, 0)   # 0
p arr.fetch(10, nil) # nil
```

Метод `fetch` может принимать блок, внутри которого можно задать поведение метода при выходе за границу массива (листинг 23.14).

Листинг 23.14. Использование блока совместно с методом `fetch`. Файл fetch_block.rb

```
arr = [1, 2, 3, 4, 5]

puts arr.fetch(10) { |i| "Индекс #{i} не существует" }
puts arr.fetch(10) { |i| 0 }
```

Для извлечения среза массива предназначен метод `slice`. Метод может принимать один аргумент — в этом случае он ведет себя точно так же, как метод `at`, т. е. просто ищет элемент по его индексу. Метод `slice` может принимать диапазон или два числовых аргумента — это эквивалентно вызову квадратных скобок с диапазоном или двумя числовыми параметрами (листинг 23.15).

Листинг 23.15. Использование метода `slice`. Файл slice.rb

```
arr = [1, 2, 3, 4, 5]

p arr.slice(1..3)  # [2, 3, 4]
p arr.slice(1...3) # [2, 3]
p arr.slice(1..-2) # [2, 3, 4]
```

```
p arr.slice(1, 1)  # [2]
p arr.slice(2, 1)  # [3]
p arr.slice(2, 2)  # [3, 4]
p arr.slice(-2, 2) # [4, 5]
```

Метод `slice` возвращает новый массив, не подвергая изменению исходный. Если необходимо заменить оригинальный массив срезом, следует прибегнуть к bang-варианту метода `slice!` (листинг 23.16).

Листинг 23.16. Замена оригинала срезом массива. Файл slice_bang.rb

```
arr = [1, 2, 3, 4, 5]

p arr.slice(2..3)  # [3, 4]
p arr              # [1, 2, 3, 4, 5]
p arr.slice!(2..3) # [3, 4]
p arr              # [1, 2, 5]
```

Метод `values_at` позволяет извлечь элементы с произвольными индексами. В листинге 23.17 заводится массив из семи элементов и при помощи метода `values_at` извлекаются элементы с индексами `1`, `3` и `5`.

Листинг 23.17. Замена оригинала срезом массива. Файл values_at.rb

```
arr = (1..7).to_a
p arr.values_at(1, 3, 5) # [2, 4, 6]
```

Извлечение первого и последнего элемента — это частые операции, поэтому для них предусмотрены отдельные методы: `first` и `last` (листинг 23.18).

Листинг 23.18. Извлечение первого и последнего элементов. Файл first_last.rb

```
arr = [1, 2, 3, 4, 5]

p arr.first # 1
p arr.last  # 5
```

Для извлечения первых нескольких элементов можно воспользоваться методом `take`. Метод принимает в качестве аргумента целое число — количество извлекаемых элементов (листинг 23.19).

Листинг 23.19. Извлечение первых нескольких элементов. Файл take.rb

```
arr = [1, 2, 3, 4, 5]
p arr.take(3) # [1, 2, 3]
```

Метод `drop` исключает первые несколько элементов из начала массива и возвращает получившийся остаток (листинг 23.20).

Листинг 23.20. Исключение первых нескольких элементов. Файл drop.rb

```
arr = [1, 2, 3, 4, 5]
p arr.drop(3) # [4, 5]
```

На рис. 23.6 приводится схема работы методов `take` и `drop`. Оба метода формируют новые массивы, исходный массив `arr` остается неизменным.

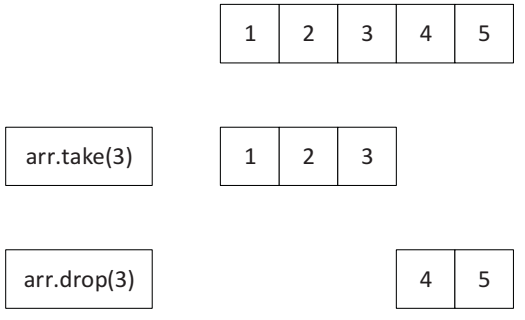


Рис. 23.6. Схема работы методов `take` и `drop`

Методы `take` и `drop` позволяют разбить массив по индексу и получить первую и вторую половины разделенного массива. Методы `take_while` и `drop_while` действуют похожим образом. Однако вместо числового параметра они принимают блок, в котором задается условие по разбивке массива.

Метод `take_while` возвращает подмассив от начала исходного массива до первого элемента, для которого блок вернет ложное значение (`false`). Метод `drop_while` извлекает «хвост» массива, начиная с элемента, для которого блок вернул `false`, и до конца массива (листинг 23.21).

Листинг 23.21. Файл take_drop_while.rb

```
arr = [1, 2, 3, 4, 5]

p arr.take(3) # [1, 2, 3]
p arr.drop(3) # [4, 5]

p arr.take_while { |x| x < 3 } # [1, 2]
p arr.drop_while { |x| x < 3 } # [3, 4, 5]

arr = [1, 3, 5, 4, 2]

p arr.take_while { |x| x < 3 } # [1]
p arr.drop_while { |x| x < 3 } # [3, 5, 4, 2]
```

23.4. Поиск индексов элементов

При помощи метода `index` по значению элемента массива можно найти его индекс. Если искомых значений в массиве несколько, возвращается индекс первого найденного элемента. Метод `rindex` действует аналогично методу `index`, однако ищет с конца массива (листинг 23.22).

Листинг 23.22. Поиск индекса элемента. Файл index.rb

```
colors = %w[red orange yellow green red blue red indigo red violet]

p colors.index('red')   # 0
p colors.index('hello') # nil
p colors.rindex('red')  # 8
```

Если элемент не обнаружен, методы `index` и `rindex` возвращают неопределенное значение `nil`. По умолчанию сравнение производится при помощи оператора ==. Логику поиска можно изменить, если вместо аргумента передать методу блок (листинг 23.23). В этом случае метод возвратит первый элемент, для которого блок вернет истинное значение (`true`).

Замечание

Метод `index` имеет синоним `find_index`.

Листинг 23.23. Передача блока методу `index`. Файл index_block.rb

```
arr = [-3, -2, -1, 0, 1, 2, 3]
puts arr.index { |value| value > 0 } # 4
```

23.5. Случайный элемент массива

Для получения случайного элемента массива можно воспользоваться уже знакомым нам методом `rand`. Метод принимает в качестве аргумента диапазон и возвращает случайное значение из этого диапазона (листинг 23.24).

Листинг 23.24. Получение случайного элемента массива. Файл rand.rb

```
colors = %w[red orange yellow green blue indigo violet]

index = rand(0..6)

puts index         # 2
puts colors[index] # yellow
```

Прием вычисления случайного индекса массива — это самый быстрый способ получения случайного элемента.

Еще одним способом получения случайного значения массива является метод `sample`. Метод может принимать в качестве аргумента число — именно такое количество случайных элементов будет извлечено из массива (листинг 23.25).

Листинг 23.25. Использование метода `sample`. Файл sample.rb

```
arr = [1, 2, 3, 4, 5]

p arr.sample    # 5
p arr.sample(2) # [4, 1]
```

Метод `shuffle` возвращает новый массив, в котором элементы оригинального массива перемешаны в случайном порядке. Bang-вариант метода `shuffle!` изменяет оригинальный массив (листинг 23.26).

Листинг 23.26. Перемешивание элементов массива. Файл shuffle.rb

```
arr = [1, 2, 3, 4, 5]

p arr.shuffle  # [4, 2, 1, 5, 3]
p arr          # [1, 2, 3, 4, 5]
p arr.shuffle! # [2, 1, 4, 5, 3]
p arr          # [2, 1, 4, 5, 3]
```

23.6. Удаление элементов

Удалять элементы из конца массива можно при помощи метода `pop`. Метод возвращает удаленный элемент, а сам массив укорачивается (листинг 23.27).

Листинг 23.27. Удаление элементов из конца массива. Файл pop.rb

```
arr = [*1..5]

p arr.pop # 5
p arr.pop # 4

p arr     # [1, 2, 3]
```

Метод `pop` может принимать необязательный числовой аргумент — количество удаляемых элементов. Если аргумент задан, метод возвращает массив, даже если удаляется лишь один элемент (листинг 23.28).

Листинг 23.28. Использование метода pop с аргументом. Файл pop_n.rb

```
arr = [*1..5]

p arr.pop(1) # [5]
p arr.pop(2) # [3, 4]

p arr        # [1, 2]
```

Удалять элементы из начала массива можно при помощи метода `shift` (листинг 23.29). Его действие аналогично методу `pop` — по умолчанию удаляется один элемент, который возвращается в качестве результата.

Листинг 23.29. Удаление элементов из начала массива. Файл shift.rb

```
arr = [*1..7]

p arr.shift    # 1
p arr.shift    # 2

p arr          # [3, 4, 5, 6, 7]

p arr.shift(1) # [3]
p arr.shift(2) # [4, 5]

p arr          # [6, 7]
```

Метод `shift` может принимать необязательный параметр — количество удаляемых элементов. В этом случае в качестве результата возвращается массив удаленных элементов.

Метод `delete` позволяет удалить элементы по значению. В листинге 23.30 из массива удаляются все элементы со значением `2`.

Листинг 23.30. Удаление элементов по значению. Файл delete.rb

```
arr = [1, 2, 3, 2, 4, 2, 5]

p arr.delete(2)                              # 2
p arr                                        # [1, 3, 4, 5]

p arr.delete(10)                             # nil
p arr                                        # [1, 3, 4, 5]

p arr.delete(10) { 'индекс не обнаружен' } # "индекс не обнаружен"
```

Если методу `delete` передается индекс несуществующего элемента, массив не подвергается изменению, а метод возвращает значение `nil`. Вместо `nil` в блоке метода можно задать значение по умолчанию.

Для удаления элемента с определенным индексом предназначен метод `delete_at` (листинг 23.31).

Листинг 23.31. Удаление элементов по индексу. Файл delete_at.rb

```
arr = [1, 2, 3, 4, 5]

p arr.delete_at(2)   # 3
p arr.delete_at(100) # nil
p arr                # [1, 2, 4, 5]
```

Метод `delete_at` возвращает удаленный элемент. В случае, если элемент с таким индексом не обнаружен, — возвращается значение `nil`.

Удалять элементы можно в том числе и при помощи оператора `[]=`. Чтобы удалить элементы, достаточно присвоить диапазону пустой массив:

```
 > arr = [*1..5]
=> [1, 2, 3, 4, 5]
 > arr[1..2]
=> [2, 3]
 > arr[1..2] = []
=> []
 > arr
=> [1, 4, 5]
```

Вариант оператора `[]=` с двумя параметрами тоже позволяет удалить элементы:

```
 > arr = [*1..5]
=> [1, 2, 3, 4, 5]
 > arr[2,3]
=> [3, 4, 5]
 > arr[2,3] = []
=> []
 > arr
=> [1, 2]
```

Полностью удалить элементы массива можно при помощи метода `clear` (листинг 23.32).

Листинг 23.32. Удаление всех элементов массива. Файл clear.rb

```
arr = [1, 2, 3, 4, 5]
arr.clear
p arr # []
```

При помощи метода `compact` можно удалить из массива все неопределенные значения `nil`. Метод имеет bang-вариант `compact!`, который преобразует исходный массив (листинг 23.33).

Листинг 23.33. Удаление неопределенных значений массива. Файл compact.rb

```
arr = [1, 2, 3]
arr[7] = 4
p arr          # [1, 2, 3, nil, nil, nil, nil, 4]
p arr.compact  # [1, 2, 3, 4]
p arr          # [1, 2, 3, nil, nil, nil, nil, 4]
p arr.compact! # [1, 2, 3, 4]
p arr          # [1, 2, 3, 4]
```

Метод `uniq` позволяет получить копию массива, в котором удалены все дублирующие элементы. Bang-вариант метода `uniq!` преобразует исходный массив (листинг 23.34).

Листинг 23.34. Получение уникальных элементов массива. Файл uniq.rb

```
arr = [1, 2, 1, 3, 1, 2, 4]
p arr.uniq  # [1, 2, 3, 4]
p arr       # [1, 2, 1, 3, 1, 2, 4]
p arr.uniq! # [1, 2, 3, 4]
p arr       # [1, 2, 3, 4]
```

23.7. Замена элементов

Ruby предоставляет несколько механизмов для замены элементов массива. Самый компактный из них — это оператор `[]=`:

```
 > arr = [*1..5]
=> [1, 2, 3, 4, 5]
 > arr[1..2]
=> [2, 3]
 > arr[1..2] = [6, 7]
=> [6, 7]
 > arr
=> [1, 6, 7, 4, 5]
 > arr[1..2] = [8, 9, 10]
=> [8, 9, 10]
 > arr
=> [1, 8, 9, 10, 4, 5]
```

Метод `fill` позволяет заполнить массив значениями. В простейшем случае метод принимает единственный аргумент, которым заполняется массив (листинг 23.35).

Листинг 23.35. Заполнение массива элементами. Файл fill.rb

```
arr = Array.new(5)
p arr          # [nil, nil, nil, nil, nil]
```

```
p arr.fill(0)       # [0, 0, 0, 0, 0]
p arr.fill { |i| i + 1 } # [1, 2, 3, 4, 5]
```

Метод `fill` может принимать блок, через параметр которого передается текущий индекс. Кроме того, метод может принимать дополнительные параметры, которые задают интервал преобразований: один дополнительный параметр позволяет задать диапазон индексов, а два параметра задают начальный индекс и количество элементов, которые необходимо преобразовать (листинг 23.36).

Листинг 23.36. Дополнительные параметры метода `fill`. Файл fill_additional.rb

```
arr = Array.new(5)

p arr                          # [nil, nil, nil, nil, nil]
p arr.fill(0, 2..3)            # [nil, nil, 0, 0, nil]
p arr.fill(0, 1, 3)            # [nil, 0, 0, 0, nil]

p arr.fill(nil)                # [nil, nil, nil, nil, nil]
p arr.fill(2..3) { |i| i + 1 } # [nil, nil, 3, 4, nil]
p arr.fill(1, 3) { |i| i + 1 } # [nil, 2, 3, 4, nil]
```

Метод `replace` заменяет массив другим массивом (листинг 23.37).

Листинг 23.37. Замена содержимого массива методом `replace`. Файл replace.rb

```
arr = [1, 2, 3, 4, 5]
arr.replace %w[a b c]
p arr # ["a", "b", "c"]
```

Метод `reverse` меняет порядок элементов массива на обратный. Bang-вариант метода `reverse!` изменяет исходный массив (листинг 23.38).

Листинг 23.38. Переворачивание массива. Файл reverse.rb

```
arr = [1, 2, 3, 4, 5]
p arr.reverse  # [5, 4, 3, 2, 1]
p arr          # [1, 2, 3, 4, 5]
p arr.reverse! # [5, 4, 3, 2, 1]
p arr          # [5, 4, 3, 2, 1]
```

Метод `rotate` позволяет сдвинуть элементы массива. По умолчанию элементы сдвигаются влево на одну позицию. Уходящие за левую границу первые элементы размещаются на освобождающихся местах в конце массива.

Если методу в качестве аргумента передать отрицательное значение, элементы сдвигаются вправо. Значение аргумента задает число позиций, на которое сдвигаются элементы массива. Метод `rotate` возвращает копию массива, bang-вариант `rotate!` осуществляет преобразование в оригинальном массиве (листинг 23.39).

Листинг 23.39. Сдвиг элементов методом `rotate`. Файл rotate.rb

```
arr = [1, 2, 3, 4, 5]
p arr.rotate!     # [2, 3, 4, 5, 1]
p arr.rotate!     # [3, 4, 5, 1, 2]
p arr.rotate!(2)  # [5, 1, 2, 3, 4]

arr = [1, 2, 3, 4, 5]
p arr.rotate!(-1) # [5, 1, 2, 3, 4]
p arr.rotate!(-1) # [4, 5, 1, 2, 3]
p arr.rotate!(-2) # [2, 3, 4, 5, 1]
```

23.8. Информация о массиве

Для получения размера массива предназначен метод `length`, который имеет синоним `size` (листинг 23.40).

Листинг 23.40. Получение размера массива. Файл length.rb

```
arr = [1, 2, 3]
arr[7] = 4

p arr        # [1, 2, 3, nil, nil, nil, nil, 4]
p arr.length # 8
p arr.size   # 8
```

Метод `count` по умолчанию ведет себя точно так же, как метод `length`, однако принимает дополнительные необязательные аргументы. Если методу передать значение, он вернет количество вхождений этого объекта в массив (листинг 23.41).

Листинг 23.41. Получение размера массива. Файл count.rb

```
arr = [1, 2, 3]
arr[7] = 3

p arr            # [1, 2, 3, nil, nil, nil, nil, 4]
p arr.count      # 8
p arr.count(nil) # 4
p arr.count(3)   # 2
```

Метод `count` может принимать блок, который должен возвращать `true` или `false`. Метод подсчитывает количество элементов в массиве, для которых блок вернул истинное значение (`true`).

Для проверки, входит ли значение в массив, предназначен логический метод `include?` (листинг 23.42).

Листинг 23.42. Проверка вхождения элемента в массив. Файл include.rb

```
arr = [1, 2, 3, 4, 5]

p arr.include? 3  # true
p arr.include? 10 # false
```

Следует иметь в виду, что пустой массив в `if`-конструкции рассматривается как `true`. Поэтому, если необходимо убедиться, что массив не содержит ни одного элемента, лучше воспользоваться методом `empty?` (листинг 23.43).

Листинг 23.43. Проверка, является ли массив пустым. Файл empty.rb

```
first = []
second = [1, 2, 3, 4, 5]

puts 'Массив first пустой' if first.empty?   # true
puts 'Массив second пустой' if second.empty? # false
```

23.9. Преобразование массива

В *главе 15* мы уже рассматривали методы преобразования объектов Ruby. Массивы можно преобразовать в строку при помощи метода `join`, который применяет к каждому из элементов массива метод `to_s` и объединяет их в строку. Если методу `join` передать необязательный аргумент — он будет использоваться в качестве строки-разделителя (листинг 23.44).

Листинг 23.44. Преобразование массива в строку. Файл join.rb

```
arr = [1, 2, 3, 4, 5]

puts arr.join     # 12345
puts arr.join '-' # 1-2-3-4-5
```

Обратную операцию — преобразования строки в массив — можно выполнить при помощи метода `split` (листинг 23.45).

Листинг 23.45. Преобразование строки в массив. Файл split.rb

```
p '12345'.split ''      # ["1", "2", "3", "4", "5"]
p '1-2-3-4-5'.split '-' # ["1", "2", "3", "4", "5"]
```

Специально подготовленный массив можно преобразовать в хэш при помощи метода `to_h`. Элементы такого массива сами должны быть массивами из двух элементов: первый становится ключом, второй — значением хэша (листинг 23.46).

Листинг 23.46. Преобразование массива в хэш. Файл to_h.rb

```
colors = [
  [:red, 'красный'],
  [:orange, 'оранжевый'],
  [:yellow, 'желтый'],
  [:green, 'зеленый'],
  [:blue, 'голубой'],
  [:indigo, 'синий'],
  [:violet, 'фиолетовый'],
]

clrhsh = colors.to_h
puts clrhsh[:red]    # красный
puts clrhsh[:violet] # фиолетовый
```

Хэш можно преобразовать обратно в массив при помощи метода `to_a` (листинг 23.47).

Листинг 23.47. Преобразование хэша в массив. Файл to_a.rb

```
clrhsh = {
  red: 'красный',
  orange: 'оранжевый',
  yellow: 'желтый',
  green: 'зеленый',
  blue: 'голубой',
  indigo: 'синий',
  violet: 'фиолетовый'
}

colors = clrhsh.to_a
p colors.first # [:red, "красный"]
p colors.last  # [:violet, "фиолетовый"]
```

23.10. Арифметические операции с массивами

Складывать массивы друг с другом можно при помощи метода `concat`:

```
 > [1, 2, 3].concat([2, 3, 4])
=> [1, 2, 3, 2, 3, 4]
```

Как видим, результирующий массив содержит элементы обоих массивов. Вместо метода `concat` можно использовать оператор `+`:

```
 > [1, 2, 3] + [2, 3, 4]
=> [1, 2, 3, 2, 3, 4]
```

Поддерживаются и другие операторы — например, вычитание:

```
 > [1, 2, 3] - [2, 3, 4]
=> [1]
```

Если при сложении массивов необходимо получить только уникальные элементы, можно воспользоваться оператором объединения `|`:

```
 > [1, 2, 3] | [2, 3, 4]
=> [1, 2, 3, 4]
```

В двух этих массивах совпадают элементы `2` и `3` — при объединении массивов мы получаем только уникальные элементы. Аналогичных результатов можно добиться при помощи метода `union`:

```
 > [1, 2, 3].union [2, 3, 4]
=> [1, 2, 3, 4]
```

При помощи оператора `&` можно получать пересечение массивов. В этом случае в результирующий массив попадают элементы, которые присутствуют и в первом, и во втором массивах:

```
 > [1, 2, 3] & [2, 3, 4]
=> [2, 3]
```

23.11. Логические методы

В качестве элементов массива могут выступать логические значения истины (`true`) и лжи (`false`). Более того, любые объекты Ruby можно рассматривать как логические элементы — `false` и `nil` выступают как ложное значение, все остальные объекты рассматриваются как истина.

Ruby предоставляет ряд методов, которые облегчают проверку логического состояния массива. Например, при помощи метода `all?` можно проверить, являются ли все элементы массива истинными (листинг 23.48).

Листинг 23.48. Проверка на истинность элементов массива. Файл all.rb

```
puts [true, true, true].all?  # true
puts [true, false, true].all? # false
puts [1, 2, 3, 4, 5].all?     # true
puts [1, 2, nil, 4].all?      # false
```

Метод `all?` возвращает `true`, если все элементы истинны, иначе возвращает `false`.

Метод `all?` может принимать необязательный параметр. Если он передается, то параметр последовательно сравнивается с каждым элементом массива. Метод возвращает `true`, если все результаты сравнения истинны, иначе возвращается `false` (листинг 23.49).

Листинг 23.49. Использование необязательного аргумента `all?`. Файл all_param.rb

```
puts Array.new(5, 2).all?(2)       # true
puts [1, 2, 3, 4, 5].all?(2)       # false
puts [1, 2, 3, 4, 5].all?(Integer) # true
puts [nil, true, 2].all?(Integer)  # false
```

Метод `all?` может принимать блок, в котором можно самостоятельно задать логику вычисления истинности. В листинге 23.50 метод используется для проверки, являются ли элементы массива положительными. Если блок для каждого из элементов возвращает `true` — метод возвращает `true`, если хотя бы для одного из элементов блок вернул `false` — метод вернет `false`.

Листинг 23.50. Все ли элементы массива больше нуля? Файл all_block.rb

```
puts [1, 2, 3, 4, 5].all? { |x| x > 0 }          # true
puts [1, 2, 3, 4, 5].all? { |x| x.positive? }    # true
puts [1, 2, 3, 4, 5].all?(&:positive?)           # true

puts [-2, -1 , 0, 1, 2].all? { |x| x > 0 }       # false
puts [-2, -1 , 0, 1, 2].all? { |x| x.positive? } # false
puts [-2, -1 , 0, 1, 2].all?(&:positive?)        # false
```

Метод `any?` действует иначе — он возвращает `true`, если хотя бы один элемент массива является истинным, иначе возвращается `false` (листинг 23.51).

Листинг 23.51. Использование метода `any?`. Файл any.rb

```
puts [false, false, false].any? # false
puts [false, false, true].any?  # true
puts Array.new(5).any?          # false
puts [1, 2, 3, 4, 5].any?       # true
```

При передаче методу `any?` аргумента каждый элемент массива сравнивается с ним. Если хотя бы одно из сравнений является истинным, метод возвращает `true`, иначе возвращается `false` (листинг 23.52).

Листинг 23.52. Использование необязательного аргумента `any?`. Файл any_param.rb

```
puts [1, 2, 3, 4, 5].any?(6)           # false
puts [1, 2, 3, 4, 5].any?(2)           # true
puts [true, false, true].any?(Integer) # false
puts [nil, true, 2].any?(Integer)      # true
```

Метод `any?` также может принимать блок, в котором можно задать собственную логику вычисления истинности для элемента. В листинге 23.53 метод `any?` возвращает `true`, если хотя бы один из элементов массива является четным.

Листинг 23.53. Есть ли среди элементов массива четное значение? Файл any_block.rb

```
puts [1, 3, 5, 7, 9].any? { |x| x.even? } # false
puts [1, 3, 5, 7, 9].any?(&:even?)        # false
puts [1, 2, 3, 4, 5].any? { |x| x.even? } # true
puts [1, 2, 3, 4, 5].any?(&:even?)        # true
```

Метод `one?` проверяет, входит ли аргумент в массив лишь один раз. При этом допускается использование блока (листинг 23.54). Если аргумент не задан, метод проверяет, есть ли среди элементов только одно истинное выражение.

Листинг 23.54. Использование метода one?. Файл one.rb

```
puts [true, false, false].one?           # true
puts [true, false, true].one?            # false
puts [1, 2, 3, 4, 5].one?(3)             # true
puts [1, 2, 3, 4, 5].one?(6)             # false
puts [0, nil, true, false].one?(Integer) # true
puts [nil, true, false].one?(Integer)    # false
puts [-2, -1, 0, 1, 2].one?(&:negative?) # false
puts [-1, 0, 1, 2, 3].one?(&:negative?)  # true
puts [1, 2, 3, 4, 5].one?(&:negative?)   # false
```

Метод `none?` проверяет, входит ли аргумент в массив лишь один раз. Если аргумент не задан, метод проверяет, все ли элементы массива либо `nil`, либо `false`. В этом случае `none?` возвращает `true`. Если хотя бы один элемент массива отличен от `nil` или `false`, метод возвращает `false` (листинг 23.55).

Листинг 23.55. Использование метода none?. Файл none.rb

```
puts [false, false, false].none?          # true
puts [false, false, true].none?           # false
puts Array.new(5).none?                   # true
puts [1, 2, 3, 4, 5].none?                # false
puts [1, 2, 3, 4, 5].none?(6)             # true
puts [1, 2, 3, 4, 5].none?(3)             # false
puts [nil, true, false].none?(Integer)    # true
puts [0, nil, true, false].none?(Integer) # false
puts [1, 2, 3, 4, 5].none?(&:negative?)   # true
puts [-1, 0, 1, 2, 3].none?(&:negative?)  # false
```

23.12. Вложенные массивы

В качестве элементов массива могут выступать в том числе и массивы. В этом случае говорят о *вложенных*, или *многомерных массивах* (листинг 23.56).

Листинг 23.56. Вложенный массив. Файл array_nested.rb

```
arr = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]

p arr.first # [1, 2, 3]
p arr[1]    # [4, 5, 6]
p arr.last  # [7, 8, 9]
```

Получить вложенный массив можно, добавив в массив другой массив (листинг 23.57).

Листинг 23.57. Создание вложенного массива. Файл array_nested_create.rb

```
arr = [1, 2, 3, 4]
arr << [5, 6, 7]
p arr # [1, 2, 3, 4, [5, 6, 7]]
```

Пятый элемент массива `arr` является не числом, а массивом. Если необходимо преобразовать такой вложенный массив в линейный, можно воспользоваться методом `flatten` (листинг 23.58).

Листинг 23.58. Преобразование вложенного массива в линейный. Файл flatten.rb

```
arr = [1, 2, 3, 4]
arr << [5, 6, 7]

p arr         # [1, 2, 3, 4, [5, 6, 7]]
p arr.flatten # [1, 2, 3, 4, 5, 6, 7]
```

Для метода `flatten` предусмотрен bang-вариант `flatten!` , который преобразует исходный массив.

В случае вложенных массивов для доступа к вложенному элементу можно использовать квадратные скобки (листинг 23.59).

Листинг 23.59. Доступ к вложенным элементам массива. Файл array_nested_bracket.rb

```
arr = [1, [2, [3, 4]]]

puts arr[1][1][0] # 3
puts arr[1][1][1] # 4
```

Для этих же целей можно использовать метод `dig` (листинг 23.60).

Листинг 23.60. Использование метода `dig`. Файл dig.rb

```
arr = [1, [2, [3, 4]]]

puts arr.dig(1, 1, 0) # 3
puts arr.dig(1, 1, 1) # 4
```

Вложенные массивы удобны для представления матриц. Поэтому класс `Array` предоставляет метод `transponse`, позволяющий выполнить операцию транспонирования матрицы (листинг 23.61).

> ***Замечание***
>
> Для операции с матрицами Ruby предоставляет стандартный класс `Matrix`, рассмотрение которого выходит за рамки этой книги.

Листинг 23.61. Транспонирование матрицы. Файл transponse.rb

```
matrix = [
  [1, 2, 3],
  [4, 5, 6]
]
p matrix.transpose # [[1, 4], [2, 5], [3, 6]]
```

Метод `zip` позволяет комбинировать массивы друг с другом. В листинге 23.62 приводится пример объединения двух массивов цветов: с английскими `eng` и русскими `rus` названиями.

Листинг 23.62. Комбинирование двух массивов в один. Файл zip.rb

```
eng = %i[red orange yellow green blue indigo violet]
rus = %w[красный оранжевый желтый зеленый
            голубой синий фиолетовый]
p eng.zip(rus)
```

Результатом работы программы будет следующий массив:

```
[[:red, "красный"], [:orange, "оранжевый"], [:yellow, "желтый"], [:green,
"зеленый"], [:blue, "голубой"], [:indigo, "синий"], [:violet, "фиолетовый"]]
```

Метод `zip` формирует здесь массив, каждый элемент которого представляет собой массив из двух элементов: английского названия цвета и соответствующего ему русского названия.

Метод `zip` удобно использовать для подготовки массивов-заготовок, которые потом трансформируются в хэш при помощи метода `to_h` (см. *разд. 23.9*).

23.13. Итераторы

Все стандартные итераторы: `each`, `map`, `select`, `reject`, `reduce` и их синонимы — могут использоваться совместно с массивами (листинг 23.63).

Замечание

Итераторы уже рассматривались в *главе 11*.

Листинг 23.63. Итераторы. Файл iterators.rb

```
arr = [1, 2, 3, 4, 5]

arr.each { |x| puts x }
p arr.map { |x| x * x }      # [1, 4, 9, 16, 25]
p arr.select { |x| x.even? } # [2, 4]
p arr.reject(&:even?)        # [1, 3, 5]
p arr.reduce(:*)             # 120
```

По умолчанию итераторы `map`, `select`, `reject` и `reduce` не изменяют исходный массив. Если необходимо преобразовать оригинальную коллекцию, следует использовать bang-вариант итераторов.

Массивы — это отсортированная коллекция, в них элементы располагаются в определенном порядке. Поэтому порядок элементов, которые возвращают итераторы, каждый раз остается неизменным и соответствует порядку добавления элементов в массив.

Помимо рассмотренных ранее итераторов, класс `Array` и модуль `Enumerable` предоставляют еще несколько.

Например, для обхода массива в обратном порядке можно использовать итератор `reverse_each` (листинг 23.64).

Листинг 23.64. Обход массива в обратном порядке. Файл reverse_each.rb

```
arr = [1, 2, 3, 4, 5]

arr.reverse_each { |x| puts x }
arr.reverse.each { |x| puts x }
```

Использование итератора аналогично последовательному вызову методов `reverse` и `each`.

Итератор `each` для каждого из элементов выполняет блок только один раз. Можно использовать итератор `cycle` для того, чтобы выполнить блок многократно. Итератор принимает в качестве параметра целое число — количество повторных вызовов блока для каждого из элементов (листинг 23.65). Если параметр не указан, `cycle` будет выполняться бесконечно.

Листинг 23.65. Многократное выполнение блока для элемента массива. Файл cycle.rb

```
arr = [1, 2, 3]
arr.cycle(2) { |x| puts x } # 1, 2, 3, 1, 2, 3
```

Для получения индексов массива внутри блока можно использовать либо итератор `each_with_index`, либо комбинацию итераторов `map` и `with_index` (листинг 23.66).

Листинг 23.66. Получение индексов массива внутри блоков. Файл with_index.rb

```
arr = %w[first second first]

arr.each_with_index { |v, i| p [i, v] }
p arr.map.with_index { |v, i| [i, v] }
```

Результатом выполнения программы будут следующие строки:

```
[0, "first"]
[1, "second"]
[2, "first"]
[[0, "first"], [1, "second"], [2, "first"]]
```

Массивы допускается разбивать на подмассивы, для этого можно воспользоваться методом `each_slice`. В листинге 23.67 массив `arr` разбивается на группы из семи элементов.

Листинг 23.67. Разбивка массива на подмассивы. Файл each_slice.rb

```
arr = [*1..21]
arr.each_slice(7) { |a| p a }
```

Результатом выполнения программы будут следующие строки:

```
[1, 2, 3, 4, 5, 6, 7]
[8, 9, 10, 11, 12, 13, 14]
[15, 16, 17, 18, 19, 20, 21]
```

При помощи итератора `reduce` или его синонима `inject` можно сворачивать массив в одно значение: сумму или произведение его элементов (листинг 23.68).

Листинг 23.68. Сумма и произведение элементов массива. Файл reduce.rb

```
arr = [1, 2, 3, 4, 5]

puts arr.reduce(:*) # 120
puts arr.reduce(:+) # 15
```

Итератор `reduce` перегружен соглашениями, и чтобы расшифровать его, иногда требуются значительные усилия. Поэтому для ряда операций предоставляются спе-

циальные методы. Например, получить сумму элементов можно при помощи метода `sum` (листинг 23.69).

Листинг 23.69. Использование метода `sum`. Файл sum.rb

```
puts [1, 2, 3, 4, 5].sum # 15
```

При помощи метода `max` можно получать максимальный элемент массива, при помощи метода `min` — минимальный. Ruby предоставляет метод `minmax`, который позволяет получить минимальное и максимальное значения за один раз (листинг 23.70).

Листинг 23.70. Получение минимального и максимального значений. Файл minmax.rb

```
p [5, 8, 3, 2, 9, 1].min    # 1
p [5, 8, 3, 2, 9, 1].max    # 9
p [5, 8, 3, 2, 9, 1].minmax # [1, 9]
```

Все три метода могут получать блок, в которых можно переопределить правила сравнения элементов массива. Пусть имеется массив строк — в листинге 23.71 в качестве критерия сравнения выбирается максимальная длина строки.

Листинг 23.71. Использование блока совместно с методом `max`. Файл max_block.rb

```
puts %w[first second third].max { |a, b| a.size <=> b.size } # second
```

При поиске максимального значения элементы массива многократно сравниваются друг с другом. Блок в листинге 23.71 принимает два параметра (элемента) и задает логику их сравнения.

23.14. Сортировка массивов

Массивы можно сортировать при помощи метода `sort` (листинг 23.72). Как и для многих методов модуля `Enumerable`, для метода сортировки предусмотрен bang-вариант метода `sort!`.

Листинг 23.72. Сортировка массива. Файл sort.rb

```
p [5, 8, 3, 2, 9, 1].sort # [1, 2, 3, 5, 8, 9]
```

Метод `sort` может принимать блок, в котором можно изменить правила сравнения элементов массива (листинг 23.73).

Листинг 23.73. Использование блока совместно с блоком `sort`. Файл sort_block.rb

```
arr = %w[first second third]
p arr.sort { |a, b| a.size <=> b.size } # ["first", "third", "second"]
```

В блоке у каждого из элементов вызывается метод `size`, полученные результаты сравниваются при помощи оператора `<=>`. Можно сократить блок, если воспользоваться методом `sort_by` (листинг 23.74).

Листинг 23.74. Использование метода `sort_by`. Файл sort_by.rb

```
arr = %w[first second third]
p arr.sort_by { |x| x.size } # ["first", "third", "second"]
```

Задания

1. Запросите у пользователя число от 1 до 10. В зависимости от введенного числа создайте массив, в котором число находится в середине, а влево и вправо уходят убывающие последовательности. Например, если пользователь вводит число `3`, необходимо получить следующий массив `[1, 2, 3, 2, 1]`, в случае числа 4: `[1, 2, 3, 4, 3, 2, 1]` и т. п.
2. Пусть в переменной `matrix` задана квадратная матрица, например:

   ```
   matrix = [
     [1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]
   ]
   ```

   Следом называют сумму диагональных элементов. Для приведенного примера искомое значение: `1 + 5 + 9 = 15`. Разработайте метод, который вычисляет след квадратной матрицы произвольного размера и содержания.
3. Пусть есть диапазон от 1 до 9: `1...10`. Преобразуйте его в массив из трех подмассивов `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.
4. Создайте собственный вариант итератора `cycle`, назвав его `for`. Откройте класс `Array` и добейтесь, чтобы новый итератор был доступен для всех массивов:

   ```
   [1, 2, 3].cycle { |x| puts x }
   ```

5. При помощи итератора `reduce` получите из массива `[1, 2, 3, 4, 5]` новый массив, содержащий только нечетные элементы: `[1, 3, 5]`.
6. Расширьте класс `Integer` таким образом, чтобы числа отвечали на метод `to_a`. По умолчанию метод должен переводить число в двоичное представление и возвращать массив из нулей и единиц, представляющих десятичное число в двоичном формате:

   ```
   10.to_a # [1, 0, 1, 0]
   ```

   Необязательный аргумент метода `to_a` должен задавать разрядность числа. Например, для восьмеричных и шестнадцатеричных форматов метод должен возвращать следующие массивы:

   ```
   10.to_s(8) # [1, 2]
   10.to_s(16) # ['a']
   ```

ГЛАВА 24

Хэши

Файлы с исходными кодами этой главы находятся в каталоге *hashes* сопровождающего книгу электронного архива.

Хэши — это, наряду с массивами, еще одна упорядоченная коллекция. Однако устроены они сложнее массивов. Элемент такой коллекции вместо одного объекта хранит пару «ключ-значение». Вместо числового индекса обращаться к значению можно по ключу.

Точно так же, как и массивы, хэши подмешивают модуль `Enumerable` (см. *разд. 23.1*). Поэтому многие методы хэшей очень похожи на массивы, однако имеют специфику, связанную с тем, что элемент коллекции сам состоит из двух объектов.

Мы уже затрагивали работу с хэшами в *разд. 4.8*. В этой главе мы изучим их более детально.

24.1. Создание хэша

Самый простой способ создания хэша — использовать синтаксический конструктор «фигурные скобки» `{}`. Внутри них через запятую размещаются пары объектов: ключи и значения, разделенные последовательностью `=>` (листинг 24.1).

Листинг 24.1. Создание хэша. Файл hash.rb

```
first = {}
second = { 'first' => 1, 'second' => 2 }

p first  # {}
p second # {"first"=>1, "second"=>2}
```

В качестве ключа и значения могут выступать любые объекты языка Ruby, включая и другие хэши. Очень часто в качестве ключа используются символы. Для них действует специальное синтаксическое правило: двоеточие из начала символа можно перенести в конец. В этом случае не используется символ `=>` (листинг 24.2).

Листинг 24.2. Специальный синтаксис ключей-символов. Файл symbol.rb

```
first = { first: 1, second: 2 }
second = { first: :fst, second: :snd }

p first  # {:first=>1, :second=>2}
p second # {:first=>:fst, :second=>:snd}
```

Для создания хэша можно так же использовать метод `new` класса `Hash` (листинг 24.3).

Листинг 24.3. Использование конструктора хэша. Файл new.rb

```
p Hash.new # {}
```

Класс `Hash` позволяет создавать объекты при помощи оператора «квадратные скобки». Внутри них можно размещать пары «ключ-значение», массивы из двух элементов или просто четную последовательность объектов (листинг 24.4).

Листинг 24.4. Создание хэша при помощи квадратных скобок. Файл bracket.rb

```
p Hash[:first, 1, :second, 2]     # {:first=>1, :second=>2}
arr = [[:first, 1], [:second, 2]]
p Hash[arr]                       # {:first=>1, :second=>2}
p Hash[first: 1, second: 2]       # {:first=>1, :second=>2}
```

Впрочем, начиная с версии Ruby 2.1, более предпочтительным способом создания хэша из массива является метод `to_h` (листинг 24.5).

Листинг 24.5. Создание хэша из массива. Файл array_to_h.rb

```
arr = [[:first, 1], [:second, 2]]
p arr.to_h # {:first=>1, :second=>2}
```

24.2. Заполнение хэша

Добавлять новые элементы в хэш можно при помощи оператора `[]=` (листинг 24.6).

Листинг 24.6. Добавление элементов в хэш. Файл add.rb

```
h = {}

h['hello'] = 'world'
h[:hello] = 'ruby'

p h # {"hello"=>"world", :hello=>"ruby"}
```

Присваивание значения каждому новому ключу приводит к созданию нового элемента. Однако, если обратиться к уже существующему элементу, произойдет обновление значения существующего ключа (листинг 24.7).

Листинг 24.7. Обновление существующего значения. Файл replace.rb

```
h = {}

h[:hello] = 'hello'
p h # {:hello=>"hello"}

h[:hello] = 'ruby'
p h # {:hello=>"ruby"}
```

24.3. Извлечение элементов

Получить значение элемента можно по его ключу, который указывается в операторе «квадратные скобки» (листинг 24.8).

Листинг 24.8. Получение элемента массива. Файл get.rb

```
h = { first: 1, second: 2 }

p h[:first]              # 1
p h[:first] + h[:second] # 3
p h[:third]              # nil
```

Обращение к несуществующему ключу возвращает неопределенное значение `nil`. Помимо квадратных скобок, можно использовать метод `fetch` (листинг 24.9).

Листинг 24.9. Использование метода `fetch`. Файл fetch.rb

```
h = { first: 1, second: 2 }

puts h.fetch(:first) # 1
puts h.fetch(:third) # `fetch': key not found: :third (KeyError)
```

По умолчанию попытка обратиться при помощи `fetch` к несуществующему ключу завершается ошибкой. Однако, если задать второй необязательный аргумент, его значение будет возвращаться вместо неопределенного значения для несуществующих элементов (листинг 24.10).

Листинг 24.10. Значение по умолчанию для метода `fetch`. Файл fetch_default.rb

```
h = { first: 1, second: 2 }

puts h.fetch(:first, 0) # 1
puts h.fetch(:third, 0) # 0
```

Метод `fetch` может принимать блок, результат вычисления которого возвращается вместо несуществующего значения (листинг 24.11).

Листинг 24.11. Использование блока в методе `fetch`. Файл fetch_block.rb

```
h = { first: 1, second: 2 }

puts h.fetch(:first) { |x| "Ключ #{x} не существует" }
puts h.fetch(:third) { |x| "Ключ #{x} не существует" }
```

Результатом выполнения программы будут следующие строки:

```
1
Ключ third не существует
```

В случае вложенного хэша добраться до внутренних элементов можно при помощи нескольких квадратных скобок (листинг 24.12).

Листинг 24.12. Вложенные хэши. Файл nested_bracket.rb

```
settings = {
  title: 'Новости',
  paginate: {
    per_page: 30,
    max_page: 10
  }
}

p settings[:paginate][:per_page] # 30
p settings[:paginate][:max_page] # 10
p settings[:paginate][:total]    # nil
p settings[:seo][:keywords]      # undefined method `[]' for nil:NilClass
```

Последнее выражение в приведенной программе: `settings[:seo][:keywords]` — приводит к ошибке. Так как элемента с ключом `:seo` не существует, выражение `settings[:seo]` возвращает неопределенное значение `nil`. Объект `nil` не поддерживает оператор «квадратные скобки», и программа завершается ошибкой.

Избежать этой ситуации можно при помощи метода `dig`, который принимает в качестве аргументов ключи вложенного массива. Метод возвращает значение в случае успеха, иначе возвращает неопределенное значение `nil` (листинг 24.13).

Листинг 24.13. Использование метода `dig`. Файл dig.rb

```
settings = {
  title: 'Новости',
  paginate: {
    per_page: 30,
```

```
    max_page: 10
  }
}

p settings.dig(:paginate, :per_page) # 30
p settings.dig(:paginate, :total)    # nil
p settings.dig(:seo, :keywords)      # nil
```

24.4. Поиск ключа

Наибольшей производительности хэши достигают при извлечении значений по ключу. Тем не менее допускается поиск ключа по значению — такая операция менее производительна, но возможна.

Для ее выполнения можно воспользоваться методом `key`, который принимает в качестве аргумента значение, а возвращает ключ. Если значение не обнаружено, метод возвращает `nil` (листинг 24.14).

Листинг 24.14. Поиск ключа по значению. Файл key.rb

```
h = { first: 1, second: 2, 'hello' => 1 }

p h.key(1) # :first
p h.key(2) # :second
p h.key(3) # nil
```

Ключи в хэше уникальны, значения же могут повторяться. При поиске метод анализирует хэш слева направо и возвращает ключ для первого найденного значения. В приведенном примере два элемента хэша имеют значение `1`, и в качестве результата поиска возвращается ключ `:first`.

24.5. Обращение к несуществующему ключу

Обращение к несуществующим элементам хэша возвращает неопределенное значение `nil`. Однако это значение можно изменить при помощи атрибута `default` (листинг 24.15)

Листинг 24.15. Атрибут `default`. Файл default.rb

```
h = {}
p h.default # nil
p h[:hello] # nil

h.default = 42
p h.default # 42
p h[:hello] # 42
```

Значение по умолчанию для несуществующего ключа можно задать в виде аргумента метода `new` (листинг 24.16).

Листинг 24.16. Значение по умолчанию через параметр `new`. Файл default_param.rb

```
h = Hash.new(42)
p h.default # 42
p h[:hello] # 42
```

Этот подход не содержит «подводных камней», пока мы имеем дело с неизменяемыми объектами вроде целых чисел или символов. Проблемы могут возникать, когда в качестве значения по умолчанию выступают изменяемые объекты (листинг 24.17).

Листинг 24.17. Объект в качестве значения по умолчанию. Файл default_object.rb

```
h = Hash.new(Object.new)

p h[:hello]  #<Object:0x007fcbeb0eacb8>
p h[:world]  #<Object:0x007fcbeb0eacb8>
p h[:params] #<Object:0x007fcbeb0eacb8>
```

При обращении к несуществующим ключам все они ссылаются на один и тот же объект (рис. 24.1).

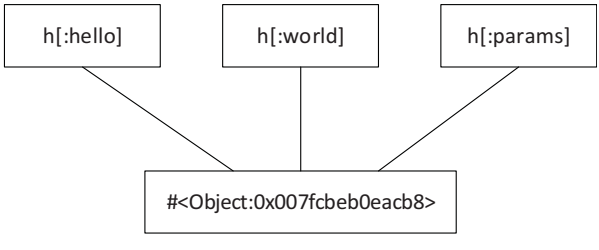


Рис. 24.1. Все элементы хэша ссылаются на один и тот же объект

Если такой изменяемый объект изменить через один ключ, изменения затронут все ключи, которые ссылаются на этот объект (листинг 24.18).

Листинг 24.18. Массив в качестве значения по умолчанию. Файл default_hash.rb

```
h = Hash.new({})

h[:params][:per_page] = 30

p h[:params]   # {:per_page=>30}
p h[:settings] # {:per_page=>30}
p h            # {}
```

Как видно из приведенного примера, несуществующему элементу `h[:params]` назначается по умолчанию хэш `{}`. После размещения в нем значения `30` с ключом `:per_page` этот элемент содержит хэш `{per_page: 30}`. Однако при обращении к любому другому несуществующему ключу мы получаем тот же самый хэш.

Гораздо чаще требуется поведение, когда любой несуществующий ключ можно рассматривать как пустой хэш. Добиться этого можно при помощи блока метода `new` (листинг 24.19).

Листинг 24.19. Массив в качестве значения по умолчанию. Файл default_block.rb

```
par = Hash.new { |h, k| h[k] = {} }

p par[:params] # {}
p par          # {:params=>{}}

par[:params][:per_page] = 30
par[:params][:max_page] = 10

par[:hello][:name] = 'Ruby'

p par # {:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}}
```

Блок вычисляется всякий раз, когда происходит обращение к несуществующему ключу хэша, — не имеет значения, происходит чтение или присваивание. Внутрь блока передаются два параметра: хэш `h` и ключ `k`. В результате каждому новому ключу можно сопоставить объект. В приведенном примере — это новый хэш.

Существует альтернативный способ задать блок Hash-объекта. Для этого можно воспользоваться атрибутом `default_proc` (листинг 24.20).

Листинг 24.20. Использование `proc`-объекта. Файл default_proc.rb

```
par = {}
par.default_proc = ->(h, k) { h[k] = {} }

par[:params][:per_page] = 30
par[:params][:max_page] = 10
par[:hello][:name] = 'Ruby'

p par # {:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}}
```

24.6. Удаление элементов хэша

Метод `slice` позволяет получить часть хэша, который содержит только те элементы, ключи которых переданы методу в качестве аргументов. Несуществующие ключи просто игнорируются (листинг 24.21).

Замечание

Метод `slice` добавлен в Ruby, начиная с версии 2.5.

Листинг 24.21. Извлечение части хэша. Файл slice.rb

```
h = { first: 1, second: 2, 'hello' => 1 }

p h.slice(:first)          # {:first=>1}
p h.slice(:first, 'hello') # {:first=>1, "hello"=>1}
p h.slice(:third, :hello)  # {}
```

Метод `slice` не имеет bang-варианта — он всегда возвращает новый хэш. Тем не менее Ruby предоставляет методы, в том числе изменяющие исходный объект хэша. Например, при помощи метода `clear` можно полностью очистить хэш (листинг 24.22).

Листинг 24.22. Удаление всех элементов хэша. Файл clear.rb

```
h = { first: 1, second: 2 }
h.clear
p h # {}
```

Для удаления элемента из начала хэша предназначен метод `shift`. Метод возвращает удаленный элемент в виде массива, содержащего ключ и значение, при этом исходный хэш становится короче (листинг 24.23).

Листинг 24.23. Удаление элементов из начала хэша. Файл shift.rb

```
h = { first: 1, second: 2}

p h       # {:first=>1, :second=>2}
p h.shift # [:first, 1]
p h       # {:second=>2}
```

Специального метода для удаления элемента из конца хэша не предусмотрено, однако можно удалить любой элемент, воспользовавшись методом `delete`. Метод принимает в качестве аргумента ключ и возвращает значение удаленного элемента. В случае, если элемент с таким ключом не обнаружен, метод возвращает неопределенное значение `nil` (листинг 24.24).

Листинг 24.24. Удаление произвольного элемента хэша. Файл delete.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h                # {:fst=>1, :snd=>2, :thd=>3, :fth=>4}
p h.delete(:fth)   # 4
p h                # {:fst=>1, :snd=>2, :thd=>3}
p h.delete(:snd)   # 2
```

```
p h                # {:fst=>1, :thd=>3}
p h.delete(:hello) # nil
p h                # {:fst=>1, :thd=>3}
```

Метод `delete` может принимать блок, в котором можно задать значение, которое будет возвращаться в случае, если ключ не обнаружен (листинг 24.25).

Листинг 24.25. Использование блока в методе `delete`. Файл delete_block.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.delete(:snd) { |e| 0 }   # 2
p h.delete(:hello) { |e| 0 } # 0
```

Метод `delete_if` позволяет удалить сразу несколько элементов. Блок метода принимает два параметра: ключ и значение, используя которые можно составить логическое выражение. Метод удаляет все элементы, для которых блок возвращает «истину» `true` (листинг 24.26).

Листинг 24.26. Удаление элементов хэша по условию. Файл delete_if.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.delete_if { |k, v| v > 2 } # {:fst=>1, :snd=>2}
p h                           # {:fst=>1, :snd=>2}
```

Как и массивы, хэши поддерживают метод `compact`, который позволяет избавиться от элементов со значением `nil`. Предусмотрен bang-вариант метода `compact!`, который преобразует исходный хэш (листинг 24.27).

> ***Замечание***
> Методы `compact` и `compact!` добавлены в Ruby, начиная с версии 2.4.

Листинг 24.27. Удаление неопределенных значений. Файл compact.rb

```
h = { fst: 1, snd: nil, thd: 3, fth: nil }

p h.compact  # {:fst=>1, :thd=>3}
p h          # {:fst=>1, :snd=>nil, :thd=>3, :fth=>nil}

p h.compact! # {:fst=>1, :thd=>3}
p h          # {:fst=>1, :thd=>3}
```

24.7. Информация о хэшах

Класс `Hash` предоставляет ряд информационных методов, позволяющих получить разнообразную информацию о хэше и его содержимом. Так как `Hash` подмешивает модуль `Enumerable` (см. *разд. 23.1*), многие методы массивов и хэшей совпадают:

```
 > Hash.ancestors
=> [Hash, Enumerable, Object, Kernel, BasicObject]
```

Например, методы `size` и `length` позволяют получить размер хэша (листинг 24.28).

Листинг 24.28. Определение размера хэша. Файл length.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

puts h.size   # 4
puts h.length # 4
```

Метод `count`, который также позволяет получить размер коллекции, имеет некоторые особенности. Например, в качестве параметра он ожидает массив из двух элементов с ключом и значением. В силу уникальности ключа результат всегда будет равен `1`, и в такой форме метод `count` практически бесполезен. Однако можно подсчитывать количество элементов хэша при помощи блока, который принимает два параметра: ключ и значение. Если блок возвращает значение `true`, метод `count` учитывает элемент в результате, если `false` — игнорирует (листинг 24.29).

Листинг 24.29. Удаление неопределенных значений. Файл count.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

puts h.count                    # 4
puts h.count([:snd, 2])         # 1
puts h.count { |_k, v| v > 2 } # 2
```

Логический метод `key?` проверяет, наличествует ли в хэше элемент с запрашиваемым ключом. Метод имеет несколько синонимов: `include?`, `has_key?` и `member?` (листинг 24.30).

Листинг 24.30. Содержит ли хэш ключ? Файл include.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.key? :thd       # true
p h.key? :hello     # false

p h.has_key? :thd   # true
p h.has_key? :hello # false

p h.include? :thd   # true
p h.include? :hello # false

p h.member? :thd    # true
p h.member? :hello  # false
```

Метод `value?` проверяет, присутствует ли в хэше элемент с запрашиваемым значением. Метод имеет синоним `has_value?` (листинг 24.31).

Листинг 24.31. Содержит ли хэш значение? Файл value.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.value? 3      # true
p h.value? 10     # false

p h.has_value? 3  # true
p h.has_value? 10 # false
```

24.8. Хэши как аргументы методов

Так как в синтаксисе хэшей и блоков могут присутствовать фигурные кавычки, очень часто возникают недоразумения при попытке использования хэша в качестве аргумента метода (листинг 24.32).

Листинг 24.32. Специальный синтаксис ключей-символов. Файл p_hash.rb

```
p { first: 1, second: 2 } # syntax error, unexpected ':', expecting '}'
```

Программа завершается не вполне очевидной ошибкой. Дело в том, что метод `p` сам может принимать блок, и Ruby-интерпретатор не в состоянии отличить его от хэша (листинг 24.33).

Листинг 24.33. Метод p может принимать блок. Файл p_block.rb

```
p(1) { } # 1
p { }  #
```

Для того чтобы поправить ситуацию, можно поместить хэш в круглые скобки. Однако более предпочтительным является удаление у хэша фигурных скобок (листинг 24.34).

Листинг 24.34. Удаление фигурных скобок у хэша-аргумента. Файл p_last_hash.rb

```
p({ first: 1, second: 2 }) # плохо
p(first: 1, second: 2)     # лучше
p first: 1, second: 2      # хорошо
```

Удалять фигурные скобки можно только у последнего хэша в списке аргументов, в остальных случаях фигурные скобки обязательны (листинг 24.35).

Листинг 24.35. Файл p_few_hashes.rb

```
p 'hello', { fst: 1, snd: 2 }, first: 1, second: 2
```

24.9. Объединение хэшей

Хэши можно объединять при помощи метода `merge`. Для метода предусмотрены bang-вариант `merge!` и синоним `update`.

В качестве аргумента метод `merge` может принимать несколько хэшей, которые последовательно применяются к хэшу-получателю. Добавляемый ключ может уже существовать в хэше, в этом случае его значение обновляется. Если такого ключа не было, он просто добавляется (листинг 24.36).

Листинг 24.36. Объединение хэшей. Файл merge.rb

```
first = { fst: 1, snd: 2 }
second = { snd: :hello, thd: :world }

p first.merge second # {:fst=>1, :snd=>:hello, :thd=>:world}
```

Как можно видеть, ключ `:snd` присутствует в обоих хэшах, однако в результирующем хэше оказалось значение из хэша-аргумента `first`. Поменять логику обновления можно при помощи блока, который принимает метод `merge` (листинг 24.37).

Листинг 24.37. Изменение правила объединения хэшей. Файл merge_block.rb

```
first = { fst: 1, snd: 2 }
second = { snd: :hello, thd: :world }

first.merge!(second) { |k, fst, snd| fst }
p first # {:fst=>1, :snd=>2, :thd=>:world}
```

24.10. Преобразование хэшей

В отношении хэшей можно использовать стандартные итераторы модуля `Enumerable`: `each`, `map`, `select`, `reject` и `reduce`. Блок итератора принимает два параметра: ключ и значение (листинг 24.38).

> ***Замечание***
> Итератор `each` имеет синоним `each_pair`.

Листинг 24.38. Итераторы. Файл iterators.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

h.each { |k, v| puts "{k} => #{v}" }
p h.map { |_k, v| v }                # [1, 2, 3, 4]
```

```
p h.select { |k, v| v > 2 }          # {:thd=>3, :fth=>4}
p h.reject { |k, v| v > 2 }          # {:fst=>1, :snd=>2}
p h.reduce(0) { |m, (k, v)| m + v } # 10
```

В блоке метода `reduce` первый параметр `m` является агрегационной переменной, а второй параметр — массивом из двух значений: ключа `k` и значения `v`. При помощи круглых скобок можно сразу разложить массив на составляющие значения и присвоить их параметрам.

Итераторы `each_key` и `each_value` позволяют обходить ключи и значения хэша. Их блоки имеют лишь один параметр (листинг 24.39).

Листинг 24.39. Методы `each_key` и `each_value`. Файл each.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

h.each_key { |k| p k.to_s }
h.each_value { |v| p v * 10 }
```

Метод `map` возвращает массив, что может быть не очень удобно, если необходимо получить хэш с измененными ключами и значениями. Для изменения значений хэша можно воспользоваться методом `transform_values` (листинг 24.40).

> ***Замечание***
>
> Метод `transform_values` добавлен в Ruby, начиная с версии 2.4.

Листинг 24.40. Преобразование значений хэша. Файл transform_values.rb

```
h = { fst: 1, snd: 2}

p h.transform_values { |v| v * 10 } # {:fst=>10, :snd=>20}
```

Для преобразования ключей хэша предназначен метод `transform_keys`. В листинге 24.41 ключи хэша `params` поочередно преобразуются к строковым и символьным значениям.

> ***Замечание***
>
> Метод `transform_keys` добавлен в Ruby, начиная с версии 2.5.

Листинг 24.41. Преобразование ключей хэша. Файл transform_keys.rb

```
params = {
  title: 'Новости',
  'page' => {
     per: 30,
    'max' => 10
  }
}
```

```
p params.transform_keys { |k| k.to_s }
# {"title"=>"Новости", "page"=>{:per=>30, "max"=>10}}
p params.transform_keys(&:to_sym)
# {:title=>"Новости", :page=>{:per=>30, "max"=>10}}
```

Для преобразования хэша в массив можно воспользоваться методом `to_a` (листинг 24.42).

Листинг 24.42. Преобразование ключей хэша. Файл to_a.rb

```
h = { fst: 1, snd: 2 }
p h.to_a # [[:fst, 1], [:snd, 2]]
```

В том случае, когда необходимо получить массив ключей или значений, удобно воспользоваться методами `keys` и `values` (листинг 24.43).

Листинг 24.43. Извлечение массива ключей и значений. Файл keys_values.rb

```
h = { fst: 1, snd: 2 }

p h.keys   # [:fst, :snd]
p h.values # [1, 2]
```

Метод `values_at` принимает произвольное количество ключей хэша и возвращает массив соответствующих им значений. Если запрашиваемый ключ не обнаружен, в результирующий массив добавляется неопределенное значение `nil` (листинг 24.44).

Листинг 24.44. Извлечение значений хэша. Файл values_at.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.values_at(:fth, :snd)   # [4, 2]
p h.values_at(:fth, :hello) # [4, nil]
```

Сходным образом действует метод `fetch_values` — он также возвращает массив значений хэша. Однако, если ключ не обнаружен, возникает ошибка. Впрочем, в отличие от `values_at`, метод может принимать блок, позволяющий задать значение для случая, когда соответствующий ключ не обнаружен (листинг 24.45).

Листинг 24.45. Использование метода `fetch_values`. Файл fetch_values.rb

```
h = { fst: 1, snd: 2, thd: 3, fth: 4 }

p h.fetch_values(:fth, :snd)              # [4, 2]
p h.fetch_values(:fth, :hello)            # key not found: :hello
p h.fetch_values(:fth, :hello) { |_k| 0 } # [4, 0]
```

Метод `to_h` позволяет получить новый хэш, в блоке можно выполнить преобразования ключа и значения для нового хэша. Например, можно поменять местами ключ и значение (листинг 24.46).

Листинг 24.46. Обмен ключей и значений в хэше. Файл to_h.rb

```
h = { fst: 1, snd: 2, thd: 1 }
p h.to_h { |k, v| [v, k]} # {1=>:thd, 2=>:snd}
```

Следует обратить внимание, что результирующий массив может содержать меньше элементов, если хэш содержит неуникальные значения.

Операцию обмена ключа и значения можно выполнить специальным методом `invert` (листинг 24.47).

Листинг 24.47. Использование метода `invert`. Файл invert.rb

```
h = { fst: 1, snd: 2, thd: 1 }
p h.invert # {1=>:thd, 2=>:snd}
```

24.11. Сравнение ключей

В качестве ключей массива могут выступать любые объекты языка Ruby. Для того чтобы глубже понимать работу хэша, следует еще раз остановиться на том, как Ruby сравнивает объекты. Обычно объекты сравниваются друг с другом при помощи оператора `==`:

```
 > 'hello' == 'hello'.dup
=> true
 > :hello == 'hello'
=> false
 > :hello == :hello
=> true
 > 2 + 2 == 4
=> true
```

По умолчанию оператор `==` считает объекты равными, если это один и тот же объект. Однако логику оператора можно изменить путем его перегрузки. В листинге 24.48 приводится пример класса `Ticket`, в котором два билета считаются равными, если у них совпадают место `seat` и ряд `row`.

Листинг 24.48. Перегрузка оператора ==. Файл ticket.rb

```
class Ticket
  attr_accessor :seat, :row

  def initialize(seat:, row:)
    @seat = seat
```

```
    @row = row
  end

  def ==(ticket)
    seat == ticket.seat && row == ticket.row
  end
end
```

Теперь два билета на один ряд и место будут считаться одинаковыми:

```
 > require_relative 'ticket'
=> true
 > fst = Ticket.new(seat: 10, row: 5)
=> #<Ticket:0x00007fb212897300 @seat=10, @row=5>
 > snd = Ticket.new(seat: 10, row: 5)
=> #<Ticket:0x00007fb212871128 @seat=10, @row=5>
 > thd = Ticket.new(seat: 2, row: 12)
=> #<Ticket:0x00007fb2138db220 @seat=2, @row=12>
 > fst == snd
=> true
 > fst == thd
=> false
```

Если нам потребуется старый способ сравнения, можно обратиться к методу `equal?`, который по-прежнему будет считать равными только те переменные, которые ссылаются на один и тот же объект:

```
 > fst.equal? snd
=> false
 > fst.equal? fst
=> true
```

Метод `equal?`, в отличие от оператора `==`, не принято перегружать — он всегда должен позволять сравнивать объекты классическим способом.

В некоторых случаях сравнение объектов осуществляется альтернативным способом. Например, внутри конструкции `case` для сравнения используется оператор `===` (см. *разд. 8.5*).

Сравнение ключей хэша — это тоже специальный случай. Для этого используется метод `hash`, который возвращает уникальный идентификатор объекта:

```
 > 'hello'.hash
=> 1540466684107775021
 > String.new('hello').hash
=> 1540466684107775021
 > 'hello'.dup.hash
=> 1540466684107775021
 > :hello.hash
=> -3441870540755448197
```

```
 > 4.hash
=> 2051630799338892664
 > (2 + 2).hash
=> 2051630799338892664
 > Object.new.hash
=> 2296263672619834512
 > Object.new.hash
=> 2089206269800948969
```

Причем для сравнения применяется не оператор ==, а специальный метод eql?. Для большинства объектов этот метод является псевдонимом оператора ==. Однако его можно переопределить для того, чтобы изменить поведение объектов в качестве ключа. Более того, для некоторых классов eql? уже переопределен на уровне интерпретатора:

```
 > 1 == 1.0
=> true
 > 1.eql? 1.0
=> false
 > {1 => 'hello', 1.0 => 'world'}
=> {1=>"hello", 1.0=>"world"}
```

Таким образом, переопределяя методы hash и eql?, мы можем влиять на поведение объектов в качестве ключей хэша. Переработаем класс билета Ticket таким образом, чтобы объекты класса можно было использовать в качестве ключей хэша. В листинг 24.49 метод hash переопределяется с тем, чтобы для объектов с одинаковым рядом и местом возвращалось одинаковое значение. Так же переопределен и метод eql?, хотя в нашем случае это не обязательно, поскольку логика работы метода не меняется.

Листинг 24.49. Переопределение методов hash и eql?. Файл ticket_hash.rb

```
class Ticket
  attr_accessor :seat, :row

  def initialize(seat:, row:)
    @seat = seat
    @row = row
  end

  def ==(ticket)
    seat == ticket.seat && row == ticket.row
  end

  def hash
    [row, seat].hash
  end
```

```
  def eql?(ticket)
    hash == ticket.hash
  end
end
```

Теперь, если использовать объекты класса `Ticket` в качестве ключей, билеты на один ряд и место будут рассматриваться как одинаковые:

```
 > require_relative 'ticket_hash'
=> true
 > fst = Ticket.new(seat: 10, row: 5)
=> #<Ticket:0x00007fb462a2b1b0 @seat=10, @row=5>
 > snd = Ticket.new(seat: 10, row: 5)
=> #<Ticket:0x00007fb462a30160 @seat=10, @row=5>
 > thd = Ticket.new(seat: 2, row: 12)
=> #<Ticket:0x00007fb4629e23e8 @seat=2, @row=12>
 > tickets = {}
=> {}
 > tickets[fst] = Time.new(2019, 5, 10, 10, 20)
=> 2019-05-10 10:20:00 +0300
 > tickets[snd] = Time.new(2019, 5, 10, 10, 20)
=> 2019-05-10 10:20:00 +0300
 > tickets[thd] = Time.new(2019, 5, 10, 10, 20)
=> 2019-05-10 10:20:00 +0300
 > tickets
=> {#<Ticket:0x00007fb462a2b1b0 @seat=10, @row=5>=>2019-05-10 10:20:00 +0300,
#<Ticket:0x00007fb4629e23e8 @seat=2, @row=12>=>2019-05-10 10:20:00 +0300}
```

24.12. Преобразование ключей хэша

Строки и символы — это разные объекты, даже если они выглядят похоже (см. *главу 4*):

```
 > 'white' == :white
=> false
 > 'white' == :white.to_s
=> true
 > 'white'.to_sym == :white
=> true
```

Поместив значение в элемент с символьным ключом, его не получится извлечь по строковому ключу, и наоборот (листинг 24.50).

Листинг 24.50. Символы и строки не взаимозаменяемы. Файл sym_vs_str.rb

```
params = {}
params[:per_page] = 30
params['max_page'] = 10
```

```
p params             # {:per_page=>30, "max_page"=>10}
p params[:per_page]  # 30
p params['max_page'] # 10
p params['per_page'] # nil
p params[:max_page]  # nil
```

Ключи хэша можно преобразовать к символам или строкам при помощи метода `transform_keys` (см. листинг 24.41). Метод `transform_keys` выполняет преобразования только для верхнего уровня — он не выполняет рекурсивный спуск по вложенным хэшам.

Исправить ситуацию можно при помощи гема `hashie`. Проще всего установить его при помощи утилиты `gem` (см. *разд. 3.6*):

```
$ gem install hashie
Fetching hashie-3.6.0.gem
Successfully installed hashie-3.6.0
Parsing documentation for hashie-3.6.0
Installing ri documentation for hashie-3.6.0
Done installing documentation for hashie after 0 seconds
1 gem installed
```

> ***Замечание***
>
> Детальное рассмотрение гема `hashie` выходит за рамки этой книги. В текущем разделе мы рассмотрим лишь часть его возможностей. Документацию к гему можно найти на его официальной странице в GitHub: **https://github.com/intridea/hashie**.

Установив гем, подключите его при помощи метода `require`. В листинге 24.51 в стандартном классе `Hash` стирается граница между строковыми и символьными ключами. Для этого в класс включаются модули гема `MergeInitializer` и `IndifferentAccess`.

Листинг 24.51. Файл hashie_sym_str.rb

```
require 'hashie'

class Hash
  include Hashie::Extensions::MergeInitializer
  include Hashie::Extensions::IndifferentAccess
end

params = {}
params[:per_page] = 30
params['max_page'] = 10

p params             # {:per_page=>30, "max_page"=>10}
p params[:per_page]  # 30
p params['max_page'] # 10
```

```
p params['per_page'] # 30
p params[:max_page]  # 10
```

Необязательно открывать стандартный класс `Hash` — можно унаследовать от него новый и подмешать модули в производный класс (листинг 24.52).

Листинг 24.52. Наследование класса от `Hash`. Файл hashie_inheritance.rb

```
require 'hashie'

class Params < Hash
  include Hashie::Extensions::MergeInitializer
  include Hashie::Extensions::IndifferentAccess
end

params = Params.new
...
```

В фреймворке Ruby on Rails класс `Hash` обладает методами `symbolize_keys` и `stringify_keys`, которые позволяют преобразовать ключи хэша к символам или строкам. В геме `hashie` для этого можно использовать классы модуля `Hashie` (листинг 24.53).

Листинг 24.53. Файл symbolize_keys.rb

```
require 'hashie'

params = {
  title: 'Новости',
  'page' => {
     per: 30,
    'max' => 10
  }
}

p Hashie.symbolize_keys(params)
# {:title=>"Новости", :page=>{:per=>30, :max=>10}}
p Hashie.stringify_keys(params)
# {"title"=>"Новости", "page"=>{"per"=>30, "max"=>10}}
```

Методы `symbolize_keys` и `stringify_keys` можно добавить на уровень каждого хэша — за счет включения модулей `StringifyKeys` и `SymbolizeKeys` (листинг 24.54).

Листинг 24.54. Файл symbolize_keys_bang.rb

```
require 'hashie'

class Hash
  include Hashie::Extensions::StringifyKeys
```

```
  include Hashie::Extensions::SymbolizeKeys
end

params = { per_page: 30, 'max_page' => 10 }

p params.symbolize_keys # {:per_page=>30, :max_page=>10}
p params.stringify_keys # {"per_page"=>30, "max_page"=>10}

params.symbolize_keys!
p params                # {:per_page=>30, :max_page=>10}

params.stringify_keys!
p params                # {"per_page"=>30, "max_page"=>10}
```

Методы `symbolize_keys` и `stringify_keys` возвращают новую копию массива, не изменяя оригинал. При помощи их bang-вариантов можно преобразовать исходный массив.

Класс `Hashie::Mash` в составе гема `hashie` позволяет обращаться к ключам, как к методам (листинг 24.55).

Листинг 24.55. Обращение к ключам массива, как к методам. Файл hashie_mash.rb

```
require 'hashie'

params = Hashie::Mash.new
params[:per_page] = 30
params['max_page'] = 10

p params.per_page # 30
p params.max_page # 10
```

Задания

1. Создайте коллекцию по численности стран Европы. Реализуйте ее в виде хэша, в котором ключом выступает название страны, а значением — численность населения.
2. Пусть имеется хэш цветов `colors`, в котором ключом выступает английское название цвета, а значением — русское:

```
colors = {
  red: 'красный',
  orange: 'оранжевый',
  yellow: 'желтый',
  green: 'зеленый',
  blue: 'голубой',
```

```
  indigo: 'синий',
  violet: 'фиолетовый'
}
```

Извлеките из хэша `colors`: 1) массив с русскими названиями цветов и 2) массив с английскими названиями цветов. Кроме того, 3) создайте новый хэш, в котором ключами выступают русские названия, а значениями — английские.

3. Пусть имеется хэш, в котором ключом выступает название книги, а значением — массив авторов:

```
authors = {
  'Design Patterns in Ruby' => ['Russ Olsen'],
  'Eloquent Ruby' => ['Russ Olsen'],
  'The Well-Grounded Rubyist' => ['David A. Black'],
  'The Ruby Programming Language' => ['David Flanagan',
                                      'Yukihiro Matsumoto'],
  'Metaprogramming Ruby 2' => ['Paolo Perrotta'],
  'Ruby Cookbook' => ['Lucas Carlson', 'Leonard Richardson'],
  'Ruby Under a Microscope' => ['Pat Shaughnessy'],
  'Ruby Performance Optimization' => ['Alexander Dymo'],
  'The Ruby Way' => ['Hal Fulton', 'Andre Arko']
}
```

Создайте из него новый хэш, в котором ключом будет выступать автор, а значением — количество книг, которое он написал. Отсортируйте авторов по количеству книг. В группе авторов, которые написали одинаковое количество книг, отсортируйте авторов по алфавиту.

ГЛАВА 25

Классы коллекций

Файлы с исходными кодами этой главы находятся в каталоге *collections* сопровождающего книгу электронного архива.

В этой главе мы продолжим изучение стандартных классов. Предыдущие две главы были посвящены наиболее часто используемым классам `Array` и `Hash`. Помимо них, язык программирования Ruby предоставляет также коллекционный класс — множество `Set`. Кроме того, для быстрого создания классов можно использовать специальные классы-конструкторы `Struct` и `OpenStruct`.

25.1. Множество *Set*

Множество `Set` отличается от массивов и хэшей тем, что может содержать только уникальные элементы. Прежде чем воспользоваться классом, потребуется подключить его при помощи метода `require` или `require_relative`.

`Set`, как и остальные коллекционные классы языка Ruby, включает модуль `Enumerable` (листинг 25.1).

Листинг 25.1. Класс `Set` включает модуль `Enumerable`. Файл ancestors.rb

```
require 'set'
p Set.ancestors # [Set, Enumerable, Object, Kernel, BasicObject]
```

Создать множество можно при помощи метода `Set.new`, который способен принимать в качестве аргумента массив (листинг 25.2).

Листинг 25.2. Создание множества. Файл set.rb

```
require 'set'

p Set.new # #<Set: {}>

workday = %w[monday tuesday wednesday thursday friday]
p Set.new workday
# #<Set: {"monday", "tuesday", "wednesday", "thursday", "friday"}>
```

Для создания объекта множества допускается использование квадратных скобок класса `Set` (листинг 25.3).

Листинг 25.3. Создание множества при помощи квадратных скобок. Файл bracket.rb

```
require 'set'
p Set['saturday', 'sunday'] # #<Set: {"saturday", "sunday"}>
```

Добавлять новые элементы во множество можно при помощи оператора << (листинг 25.4).

Листинг 25.4. Добавление элементов во множество. Файл set_add.rb

```
require 'set'

workday = Set.new

p workday # #<Set: {}>

workday << 'monday'
workday << 'tuesday'
workday << 'wednesday'
workday << 'thursday'
workday << 'friday'

p workday

# #<Set: {"monday", "tuesday", "wednesday", "thursday", "friday"}>
```

Оператор << имеет метод-синоним `add`, который ведет себя точно так же.

Успешно добавляются только новые элементы, попытка добавить уже существующий элемент игнорируется (листинг 25.5). Множество может содержать лишь уникальные элементы.

Листинг 25.5. Файл copy_ignore.rb

```
require 'set'

weekend = Set.new %w[saturday sunday]

p weekend # #<Set: {"saturday", "sunday"}>

weekend << 'saturday'
weekend << 'sunday'

p weekend # #<Set: {"saturday", "sunday"}>
```

Удалить элемент из множества можно при помощи метода `delete` (листинг 25.6).

Листинг 25.6. Удаление элементов из множества. Файл set_delete.rb

```
require 'set'

workday = Set.new %w[monday tuesday wednesday thursday friday]

p workday.delete 'tuesday'
p workday.delete 'thursday'
```

Метод `delete` удаляет элемент и возвращает текущее состояние множества:

```
#<Set: {"monday", "wednesday", "thursday", "friday"}>
#<Set: {"monday", "wednesday", "friday"}>
```

Множества можно складывать друг с другом — для этого используется оператор «плюс»: `+` (листинг 25.7).

Листинг 25.7. Сложение множеств. Файл set_plus.rb

```
require 'set'

workday = Set.new %w[monday tuesday wednesday thursday friday]
weekend = Set.new %w[saturday sunday]

week = workday + weekend

p week
```

Результатом сложения рабочих дней `workday` с выходными `weekend` будет полное множество дней недели `week`:

```
#<Set: {"monday", "tuesday", "wednesday", "thursday", "friday", "saturday",
                                                              "sunday"}>
```

Допускается и вычитание множеств — при помощи оператора «минус»: `-` (листинг 25.8).

Листинг 25.8. Вычитание множеств. Файл set_minus.rb

```
require 'set'

week = Set.new %w[monday tuesday wednesday thursday
                  friday saturday sunday]
weekend = Set.new %w[saturday sunday]

workday = week - weekend

p workday
```

Результатом выполнения программы будет следующее множество рабочих дней недели:

```
#<Set: {"monday", "tuesday", "wednesday", "thursday", "friday"}>
```

Пересечение множеств вычисляется при помощи оператора `&`. В результате пересечения двух множеств образуется новое множество, содержащее элементы, входящие в оба исходных множества (рис. 25.1).

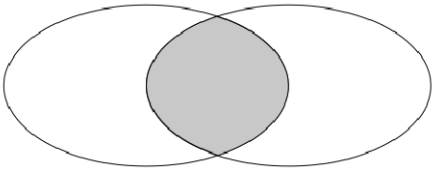


Рис. 25.1. Пересечение множеств

В листинге 25.9 приводится пример пересечения недели и выходных.

Листинг 25.9. Пересечение множеств. Файл set_intersection.rb

```
require 'set'

week = Set.new %w[monday tuesday wednesday thursday
                  friday saturday sunday]
weekend = Set.new %w[saturday sunday]

p week & weekend # #<Set: {"saturday", "sunday"}>
```

В отношении множеств допускается операция объединения кода — в результирующее множество попадают элементы обоих исходных множеств (рис. 25.2).

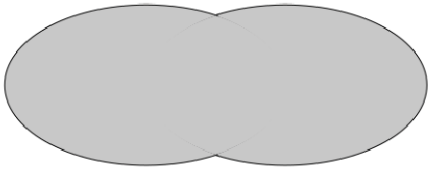


Рис. 25.2. Объединение множеств

Для объединения множества используется оператор `|` (листинг 25.10). Операторы `|` и `+` выполняют одну и ту же работу и являются синонимами.

Листинг 25.10. Объединение множеств. Файл set_union.rb

```
require 'set'

workday = Set.new %w[monday tuesday wednesday thursday friday]
weekend = Set.new %w[saturday sunday]
```

```
p workday | weekend
# #<Set: {"monday", "tuesday", "wednesday", "thursday", "friday", "saturday",
"sunday"}>
```

Исключающее пересечение множеств возвращает новое множество, в котором содержатся элементы исходных множеств, не присутствующие одновременно в обоих множествах (рис. 25.3).

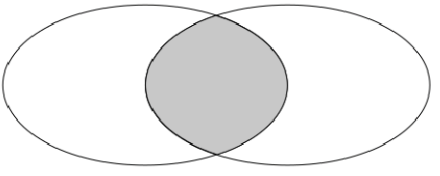


Рис. 25.3. Исключающее пересечение множеств

Для создания исключающего пересечения предназначен оператор ^ (листинг 25.11).

Листинг 25.11. Исключающее пересечение множеств. Файл set_xor.rb

```
require 'set'

fst = Set.new [1, 2, 3]
snd = Set.new [2, 3, 4]

p fst ^ snd # #<Set: {4, 1}>
```

Множества можно сравнивать друг с другом при помощи операций сравнения. Если множество входит в другое множество, оно считается меньшим:

```
 > require 'set'
=> false
 > fst = Set.new [1, 2, 3]
=> #<Set: {1, 2, 3}>
 > snd = Set.new [2, 3]
=> #<Set: {2, 3}>
 > thd = Set.new [4, 5, 6]
=> #<Set: {4, 5, 6}>
 > fst > snd
=> true
 > fst < snd
=> false
 > fst > thd
=> false
 > fst < thd
=> false
```

Если два множества не пересекаются, попытка их сравнения заканчивается ложным результатом `false`.

Множества можно преобразовать в массив при помощи метода `to_a` (листинг 25.12).

Листинг 25.12. Преобразование множества в массив. Файл set_to_a.rb

```
require 'set'
p Set['saturday', 'sunday'].to_a # ["saturday", "sunday"]
```

Так как в множества подмешан модуль `Enumerable`, их можно обходить при помощи стандартных итераторов (листинг 25.13).

Листинг 25.13. Обход множества. Файл set_iterators.rb

```
require 'set'

week = Set.new %w[monday tuesday wednesday thursday
                  friday saturday sunday]

p week.select { |d| d.start_with? 's' }.map(&:upcase)
```

В приведенной программе множество фильтруется при помощи итератора `select` — отбираются только те дни недели, которые начинаются на букву `'s'`. Далее при помощи итератора `map` и метода `upcase` отобранные дни недели переводятся в верхний регистр. Результатом выполнения программы будет следующий массив:

```
["SATURDAY", "SUNDAY"]
```

При помощи методов `length` и `size` можно узнать размер множества (листинг 25.14).

Листинг 25.14. Размер множества. Файл set_length.rb

```
require 'set'

week = Set.new %w[monday tuesday wednesday thursday
                  friday saturday sunday]
weekend = Set.new %w[saturday sunday]

p week.length           # 7
p weekend.size          # 2
p (week - weekend).size # 5
```

25.2. Класс *Struct*

Еще один класс для построения коллекций — это `Struct`. Этот класс так же включает модуль `Enumerable`:

```
 > Struct.ancestors
=> [Struct, Enumerable, Object, Kernel, BasicObject]
```

Для того чтобы продемонстрировать возможности метода `Struct`, создадим класс точки в двумерной системе координат `Point`. Класс поддерживает две инстанс-переменные: `@x` и `@y`, которые инициализируются в методе `initialize` (листинг 25.15).

Листинг 25.15. Класс точки в двумерной системе координат `Point`. Файл point.rb

```
class Point
  attr_accessor :x, :y

  def initialize(x=nil, y=nil)
    @x = x
    @y = y
  end
end
```

В листинге 25.16 создаются две точки для демонстрации работы класса `Point`.

Листинг 25.16. Использование класса `Point`. Файл point_use.rb

```
require_relative 'point'

fst = Point.new(3, 4)
p fst #<Point:0x00007fa833841358 @x=3, @y=4>

snd = Point.new
p snd #<Point:0x00007fa8338411a0 @x=nil, @y=nil>
```

Создать класс `Point` можно более простым способом при помощи `Struct` (листинг 25.17).

Листинг 25.17. Создание класса `Point` при помощи класса `Struct`. Файл struct_point.rb

```
Point = Struct.new(:x, :y)
```

Полученный класс `Point` полностью аналогичен классу из листинга 25.15. В этом можно убедиться, исправив подключаемый класс в инструкции `require_relative` (листинг 25.18).

Листинг 25.18. Файл struct_point_use.rb

```
require_relative 'struct_point'

fst = Point.new(3, 4)
p fst #<struct Point x=3, y=4>

snd = Point.new
p snd #<struct Point x=nil, y=nil>
```

Начиная с Ruby 2.5, в классах, получаемых через `Struct`, можно использовать именованные параметры. Для этого при создании класса методом `new` необходимо установить в `true` параметр `:keyword_init` (листинг 25.19).

Листинг 25.19. Файл struct_point_named.rb

```
Point = Struct.new(:x, :y, keyword_init: true)
```

После этого при создании объекта класса `Point` можно использовать именованные параметры (листинг 25.20).

Листинг 25.20. Файл struct_point_named_use.rb

```
require_relative 'struct_point_named'

p Point.new(x: 3, y: 4) #<struct Point x=3, y=4>
p Point.new(3, 4)       # wrong number of arguments (given 2, expected 0)
```

Попытка создания объекта без использования именованных параметров будет приводить к ошибке.

Объекты полученного класса `Point`, помимо методов `x` и `y`, обладают дополнительными свойствами. Например, с объектом можно обращаться как с хэшем (листинг 25.21).

Листинг 25.21. Свойства объектов класса `Point`. Файл struct_point_properties.rb

```
require_relative 'struct_point'

point = Point.new(3, 4)

puts point.x    # 3
puts point.y    # 4

puts point['x'] # 3
puts point['y'] # 4

puts point[:x]  # 3
puts point[:y]  # 4
```

Все способы доступа к атрибутам можно использовать для присваивания им значения (листинг 25.22).

Листинг 25.22. Присваивание атрибутов. Файл struct_set.rb

```
require_relative 'struct_point'

point = Point.new
p point #<struct Point x=nil, y=nil>

point.x = 3
point.y = 4
p point #<struct Point x=3, y=4>

point['x'] = 5
point['y'] = 6
p point #<struct Point x=5, y=6>

point[:x] = 7
point[:y] = 8
p point #<struct Point x=7, y=8>
```

При попытке получить доступ к несуществующему атрибуту возникает ошибка (листинг 25.23).

Листинг 25.23. Файл struct_nil.rb

```
require_relative 'struct_point'

point = Point.new(3, 4)

point[:z] = 5 # no member 'z' in struct
p point[:z]   # no member 'z' in struct
```

При помощи метода `members` можно получить список атрибутов, которые поддерживает объект (листинг 25.24).

Листинг 25.24. Список атрибутов. Файл struct_members.rb

```
require_relative 'struct_point'

point = Point.new(3, 4)
p point.members # [:x, :y]
```

Так как класс `Struct` включает модуль `Enumerable`, объекты рассматриваются как коллекции, и их можно обходить при помощи стандартных итераторов. В листинге 25.25 приводится пример обхода объекта класса `Point` при помощи итераторов `each` и `each_pair`. Блок итератора `each` принимает один параметр — значение атри-

бута. Итератор `each_pair` принимает два параметра: название атрибута и его значение.

Листинг 25.25. Обход атрибутов. Файл struct_each.rb

```
require_relative 'struct_point'

point = Point.new(3, 4)
point.each { |i| puts i }
point.each_pair { |k, v| puts "#{k} => #{v}" }
```

Результатом выполнения программы будут следующие строки:

```
3
4
x => 3
y => 4
```

Полученные из `Struct` объекты уже поддерживают сравнение — например, можно сравнить две точки (листинг 25.26).

Листинг 25.26. Сравнение объектов. Файл struct_compare.rb

```
require_relative 'struct_point'

p Point.new(3, 7) == Point.new(3, 7) # true
p Point.new(3, 7) == Point.new(7, 3) # false
```

Объекты поддерживают методы `to_a` и `to_h`, позволяющие преобразовать их к более привычным массивам и хэшам (листинг 25.27).

Листинг 25.27. Преобразование объектов к массивам и хэшам. Файл struct_to.rb

```
require_relative 'struct_point'

point = Point.new(3, 4)

p point.to_a # [3, 4]
p point.to_h # {:x=>3, :y=>4}
```

Метод `values_at` позволяет извлечь лишь часть атрибутов. В качестве параметров метод может принимать либо диапазон индексов (отсчет начинается с нуля), либо два параметра: начальный индекс и количество элементов, которые необходимо извлечь (листинг 25.28).

Листинг 25.28. Извлечение массива с частью атрибутов. Файл struct_values_at.rb

```
Point = Struct.new(:x, :y, :z)

point = Point.new(1, 2, 3)
```

```
p point.values_at(1..2) # [2, 3]
p point.values_at(0, 1) # [1, 2]
```

Struct-классы можно вкладывать друг в друга, причем доступ ко вложенным элементам можно получать как при помощи квадратных скобок, так и при помощи метода dig (листинг 25.29).

Листинг 25.29. Вложенные Struct-классы. Файл struct_dig.rb

```
Nested = Struct.new(:value)

n = Nested.new(Nested.new(Nested.new(:hello)))

p n.value.value.value           # :hello
p n[:value][:value][:value]     # :hello
p n.dig(:value, :value, :value) # :hello
p n.dig(:hello, :world)         # nil
```

При обращении к несуществующему атрибуту объекта возникает сообщение об ошибке. В то же время метод dig штатно завершается, возвращая неопределенное значение nil.

25.3. Класс *OpenStruct*

Вместо Struct на практике чаще используется класс OpenStruct, который сразу создает объект заданной структуры. Более того, OpenStruct-объекты можно расширять новыми атрибутами.

Для того чтобы воспользоваться классом OpenStruct, придется подключить библиотеку ostruct при помощи метода require (листинг 25.30).

Листинг 25.30. Использование OpenStruct. Файл ostruct.rb

```
require 'ostruct'

point = OpenStruct.new x: 3, y: 4
p point #<OpenStruct x=3, y=4>
```

Метод new класса принимает в качестве параметра хэш. В приведенном примере фигурные скобки хэша опускаются, т. к. это единственный, а значит, последний аргумент метода.

К элементам коллекции можно обращаться при помощи методов, чьи названия совпадают с названиями атрибутов (листинг 25.31).

Листинг 25.31. Получение доступа к атрибутам OpenStruct. Файл ostruct_get.rb

```
require 'ostruct'

point = OpenStruct.new x: 3, y: 4
```

```
p point.x    # 3
p point.y    # 4
p point.z    # nil

p point['x'] # 3
p point['y'] # 4
p point['z'] # nil

p point[:x]  # 3
p point[:y]  # 4
p point[:z]  # nil
```

При попытке обращения к несуществующему атрибуту объект возвращает неопределенное значение `nil`. Существующий `OpenStruct`-объект можно пополнять новыми атрибутами (листинг 25.32).

Листинг 25.32. Добавление атрибута. Файл ostruct_set.rb

```
require 'ostruct'

point = OpenStruct.new x: 3, y: 4

point.z = 5
p point #<OpenStruct x=3, y=4, z=5>
```

Классы `Struct` и `OpenStruct` очень похожи друг на друга. Разница между ними только в том, что `OpenStruct` создает сразу готовые объекты, а `Struct` — своеобразные квази-классы для создания объектов (листинг 25.33).

Листинг 25.33. Сравнение классов `OpenStruct` и `Struct`. Файл ostruct_struct.rb

```
require 'ostruct'

p Struct.new(:x, :y).new(3, 4) #<struct x=3, y=4>
p OpenStruct.new(x: 3, y: 4)   #<OpenStruct x=3, y=4>
```

Чтобы получить объект, в классе `OpenStruct` используется гораздо меньше вызовов, поэтому он встречается на практике значительно чаще, чем класс `Struct`.

Задания

1. Создайте класс `Keywords` для хранения списка ключевых слов. Каждое ключевое слово должно встречаться ровно один раз — независимо от того, в каком регистре оно добавляется в объект класса. Используйте для решения задания класс множества `Set`.

2. При помощи класса `OpenStruct` создайте классы следующих боевых кораблей:
   - атомную подводную лодку (ракеты, торпеды);
   - ракетный крейсер (ракеты);
   - военный транспорт (грузовой трюм, кран).

   Создайте по три объекта каждого из классов, снабдив случайным количеством ракет и торпед те корабли, которые могут их нести.

   Каждый из кораблей должен занимать одну клетку на игровом поле (размером 10×10). Корабли не должны занимать клетку, в которой уже находится другой корабль.
3. На шахматной доске необходимо расставить 8 ферзей таким образом, чтобы они не угрожали друг другу. Ферзь ходит по горизонтали, вертикали и диагоналям. Поэтому 8 враждебных друг другу ферзей — это максимальное их количество, которое можно разместить на шахматной доске 8×8.

   Создайте классы шахматной доски и фигур. Подсчитайте количество возможных расположений ферзей на доске.
4. Пусть имеется игровое поле 10×10 игры «Морской бой». Корабли в этой игре могут занимать 1, 2, 3 и 4 клетки — по количеству занимаемых клеток они называются однопалубными, двухпалубными, трехпалубными и четырехпалубными. Перед началом игры на поле в случайном порядке выставляется следующее количество кораблей:
   - однопалубные — 4 штуки;
   - двухпалубные — 3 штуки;
   - трехпалубные — 2 штуки;
   - четырехпалубные — 1 штука.

   Создайте классы игрового поля и кораблей. В случайном порядке расставьте корабли на игровом поле таким образом, чтобы они не занимали одни и те же клетки, а также не соприкасались границами.

ГЛАВА 26

Исключения

Файлы с исходными кодами этой главы находятся в каталоге *exceptions* сопровождающего книгу электронного архива.

В ходе выполнения программ неминуемо возникают ошибки. Это могут быть синтаксические ошибки или вызов несуществующего метода, ошибки, связанные с нехваткой памяти, сбоем внешнего сетевого сервиса, ошибки на стороне базы данных или связанные с некорректным пользовательским вводом.

В объектно-ориентированных языках программирования такие ошибки принято оформлять в виде *механизма исключений*. Предусмотреть все возможные способы обработки ошибок в классе невозможно: один и тот же класс может использоваться в составе консольной, графической программы или веб-сайта.

Вместо обработки ошибки внутри класса, предусмотрен специальный механизм передачи ошибок за его пределы — *исключения*. Разработчик класса может сгенерировать исключение, а пользователь — обработать его по своему усмотрению.

Эта глава полностью посвящена исключениям в языке Ruby.

26.1. Генерация и перехват исключений

Для того чтобы сгенерировать исключение, можно воспользоваться глобальным методом `raise`, который имеет синоним `fail` (листинг 26.1).

Листинг 26.1. Генерация исключения. Файл raise.rb

```
raise 'Ошибка'
fail 'Ошибка'
```

При возникновении ошибки штатная работа программы останавливается и происходит выход из текущих методов (рис. 26.1).

На пути исключительной ситуации могут быть размещены ее перехватчики, которые предотвращают дальнейший выход из стека вызова. Если же исключение не

встречает ни одного обработчика, оно доходит до глобальной области видимости, работа программы останавливается, а в стандартный поток выводится цепочка вызова до места возникновения ошибки и текстовое сообщение, описывающее возникшую ошибку:

```
$ ruby raise.rb
Traceback (most recent call last):
raise.rb:1:in `<main>': Ошибка (RuntimeError)
```

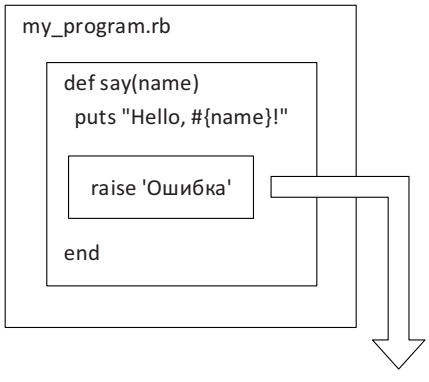


Рис. 26.1. Механизм исключительной ситуации

Исключительную ситуацию можно перехватить при помощи конструкции `begin ... rescue` (листинг 26.2).

Листинг 26.2. Перехват исключения. Файл rescue.rb

```
begin
  raise 'Ошибка'
rescue
  puts 'Произошла ошибка'
end
```

Вместо ошибки программа выведет фразу `'Произошла ошибка'`. Это означает, что исключение было перехвачено обработчиком `rescue`, исключительная ситуация не поднялась до верхнего уровня, и ошибка не возникла.

Мы можем сгенерировать исключение внутри метода класса, а перехватить его, когда используется объект этого класса. В листинге 26.3 приводится пример класса `Ticket`, в методе `initialize` которого генерируется исключение с вероятностью 50%.

Листинг 26.3. Генерация исключения по случайному закону. Файл ticket.rb

```
class Ticket
  attr_accessor :price, :date
```

```
  def initialize(date:, price: 500)
    raise 'Ошибка создания объекта' if rand(2).zero?
    @date = date
    @price = price
  end
end
```

Если при создании объекта метод `rand(2)` возвращает `0`, происходит исключительная ситуация, дальнейшее создание объекта не производится. Если вызов `rand(2)` возвращает `1`, метод `initialize` инициирует объект и штатно завершает работу.

```
 > Ticket.new date: Time.new(2019, 5, 10, 10, 20)
=> #<Ticket:0x00007fdb219f7708 @date=2019-05-10 10:20:00 +0300, @price=500>
 > Ticket.new date: Time.new(2019, 5, 10, 10, 20)
=> #<Ticket:0x00007fdb21a33b90 @date=2019-05-10 10:20:00 +0300, @price=500>
> Ticket.new date: Time.new(2019, 5, 10, 10, 20)
=> #<Ticket:0x00007fdb21910808 @date=2019-05-10 10:20:00 +0300, @price=500>
 > Ticket.new date: Time.new(2019, 5, 10, 10, 20)
Traceback (most recent call last):
        6: from /Users/i.simdyanov/.rvm/rubies/ruby-2.6.0/bin/irb:23:in
                                                                    `<main>'
        5: from /Users/i.simdyanov/.rvm/rubies/ruby-2.6.0/bin/irb:23:in `load'
        4: from /Users/i.simdyanov/.rvm/rubies/ruby-2.6.0/lib/ruby/gems/2.6.0/
                            gems/irb-1.0.0/exe/irb:11:in `<top (required)>'
        3: from (irb):11
        2: from (irb):11:in `new'
        1: from /Users/i.simdyanov/code/exceptions/ticket.rb:5:in `initialize'
RuntimeError (Ошибка создания объекта)
```

Как видим, цепочка вызовов стала гораздо длиннее, она включает в себя методы `initialize` и `new`. Для того чтобы предотвратить возникновение ошибки из-за необработанного исключения, можно воспользоваться `rescue`-обработчиком (листинг 26.4).

Листинг 26.4. Обработка исключений за пределами класса. Файл ticket_rescue.rb

```
require_relative 'ticket'

begin
  10.times.map do |i|
    p Ticket.new date: Time.new(2019, 5, 10, 10, i)
  end

  puts tickets.size
rescue
  puts 'Возникла ошибка'
end
```

Результат выполнения программы может выглядеть следующим образом:

```
#<Ticket:0x00007fa2eb858e10 @date=2019-05-10 10:00:00 +0300, @price=500>
#<Ticket:0x00007fa2eb858b90 @date=2019-05-10 10:01:00 +0300, @price=500>
#<Ticket:0x00007fa2eb8589d8 @date=2019-05-10 10:02:00 +0300, @price=500>
Возникла ошибка
```

Исключительные ситуации предназначены для внештатных ситуаций. Если пользователь программы вводит неверные данные — например, электронный адрес, в этом случае использование исключительных ситуаций оправдано. Однако, если возникает необходимость для передачи сообщения из одной части программы в другую, не следует для этого использовать механизм исключений.

26.2. Исключения — это объекты

Исключение — это объект, и когда срабатывает метод `raise` или `fail`, происходит формирование объекта исключения. В блоке `rescue` можно получить доступ к этому объекту при помощи последовательности `=>`, после которой указывается параметр (листинг 26.5).

Листинг 26.5. Доступ к объекту исключения. Файл rescue_exception.rb

```
require_relative 'ticket'

begin
  10.times.map do |i|
    p Ticket.new date: Time.new(2019, 5, 10, 10, i)
  end

  puts tickets.size
rescue => e
  p e           #<RuntimeError: Ошибка создания объекта>
  p e.class     # RuntimeError
  p e.message   # "Ошибка создания объекта"
  p e.backtrace # ["code/exceptions/ticket.rb:5:in `initialize'", ...]
end
```

У объекта исключений имеются методы. Например, получить сообщение об ошибке можно при помощи метода `message`. Метод `backtrace` возвращает массив с цепочкой вызовов до точки возникновения исключения. Метод `full_message`, добавленный, начиная с Ruby 2.6, формирует отчет из сообщения об ошибке и цепочки вызова. Этот метод возвращает строковое содержимое, которое выводится, если исключение не перехватывается обработчиком.

26.3. Стандартные ошибки

Ошибки, которые формирует интерпретатор Ruby, тоже являются исключительными ситуациями. В листинге 26.6 приводится пример программы, в которой число делится на ноль.

Листинг 26.6. Ошибка в результате деления на ноль. Файл zero_division_error.rb

```
puts 2 / 0
```

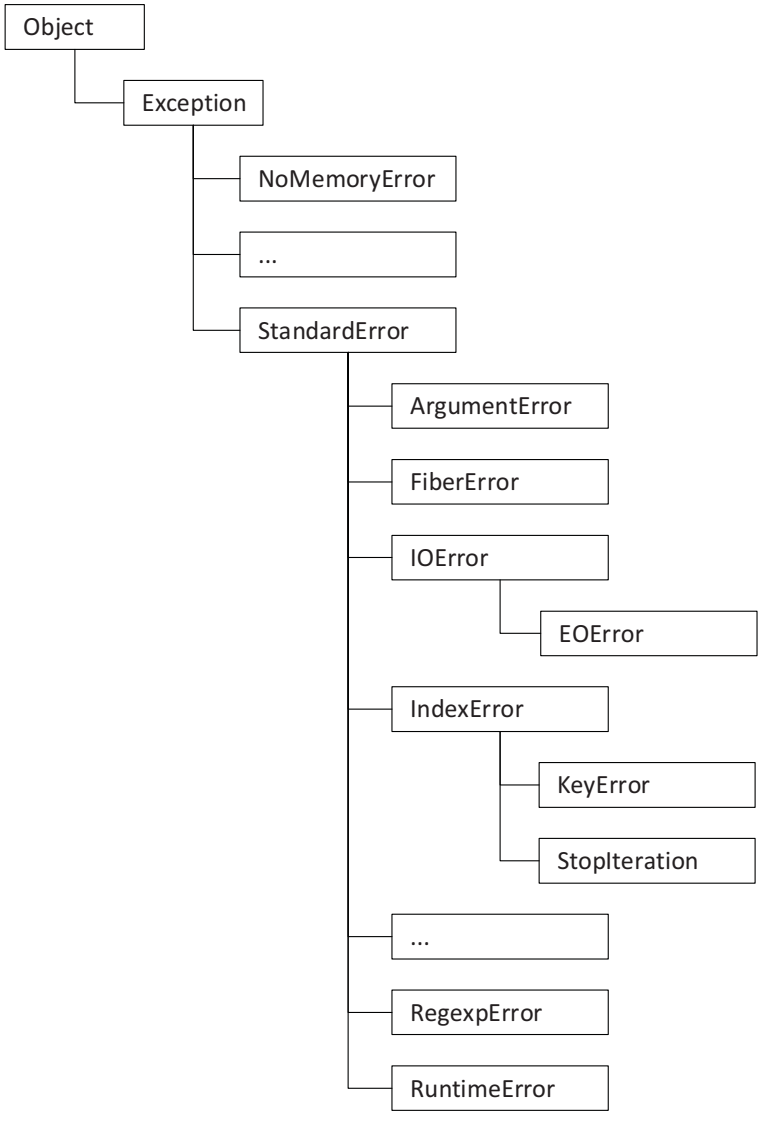


Рис. 26.2. Схема стандартных классов исключений

В арифметике деление на ноль не допускается, поэтому Ruby-интерпретатор выводит сообщение об ошибке:

```
$ ruby zero_division_error.rb
Traceback (most recent call last):
       1: from zero_division_error.rb:1:in `<main>'
zero_division_error.rb:1:in `/': divided by 0 (ZeroDivisionError)
```

Как можно видеть, ошибка — не что иное, как исключительная ситуация, за которую отвечает класс `ZeroDivisionError`. Это типичная необработанная исключительная ситуация, которую можно перехватить при помощи `rescue`-обработчика (листинг 26.7).

Листинг 26.7. Обработка ошибки деления на ноль. Файл zero_division_rescue.rb

```
begin
  puts 2 / 0
rescue => e
  puts e.message # divided by 0
end
```

Ruby предоставляет весьма обширную иерархию стандартных классов исключений, которые наследуются от базового класса `Exception` (рис. 26.2).

Если в `rescue`-блоке не указан класс ошибки, блок перехватывает только исключения, чьи классы унаследованы от `StandartError`. Остальные исключения считаются слишком серьезными, чтобы можно было что-то предпринять на уровне Ruby-программы.

26.4. Создание собственных исключений

Как видно из листинга 26.5, у объекта исключений классом является `RuntimeError`. Все классы исключений унаследованы от базового класса `Exception`:

```
 > RuntimeError.ancestors
=> [RuntimeError, StandardError, Exception, Object, Kernel, BasicObject]
```

Методы `raise` и `fail` могут принимать не только строку — первым параметром может быть указан класс или объект класса исключения. Причем объект класса можно создать как при помощи классического метода `new`, так и с помощью метода `exception`, который реализуют все классы исключений (листинг 26.8).

Листинг 26.8. Формы вызова метода `raise`. Файл raise_runtime_error.rb

```
raise 'Ошибка'
raise RuntimeError, 'Ошибка'
raise RuntimeError.new('Ошибка')
raise RuntimeError.exception('Ошибка')
```

Можно самостоятельно создавать свои собственные классы, наследуясь или непосредственно от `Exception`, или от любого производного класса (листинг 26.9).

Листинг 26.9. Создание собственного исключения. Файл rand_exception.rb

```
class RandException < Exception
end

class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    raise RandException.new('Ошибка создания объекта') if rand(2).zero?
    @date = date
    @price = price
  end
end
```

Использование собственных классов позволяет отфильтровать среди потока остальных исключений только исключения определенного класса (листинг 26.10).

Листинг 26.10. Файл rand_exception_rescue.rb

```
require_relative 'rand_exception'

begin
  10.times.map do |i|
    p Ticket.new date: Time.new(2019, 5, 10, 10, i)
  end

  puts tickets.size
rescue RandException => e
  puts "Возникла ошибка RandException: #{e.message}"
end
```

Результат запуска программы может выглядеть следующим образом:

```
#<Ticket:0x00007fa6ea0e0a90 @date=2019-05-10 10:00:00 +0300, @price=500>
Возникла ошибка RandException: Ошибка создания объекта
```

Классы исключений — это обычные классы языка Ruby, от них можно наследовать новые классы, в них можно размещать свои собственные методы, можно размещать эти классы внутри других классов и модулей.

26.5. Перехват исключений

В листинге 26.11 случайным образом генерируются исключения либо класса `RuntimeError`, либо класса `IOError`. Причем в `rescue`-блоке перехватываются только `RuntimeError`-исключения.

Листинг 26.11. Перехват исключений заданного класса. Файл runtime_rescue.rb

```
begin
  if rand(2).zero?
    raise RuntimeError, 'Ошибка класса RuntimeError'
  else
    raise IOError, 'Ошибка класса IOError'
  end
rescue RuntimeError => e
  puts e.class
  puts e.message
end
```

Последовательный запуск программы может приводить к разным результатам:

```
$ ruby runtime_rescue.rb
RuntimeError
Ошибка класса RuntimeError
$ ruby runtime_rescue.rb
Traceback (most recent call last):
runtime_rescue.rb:5:in `<main>': Ошибка класса IOError (IOError)
```

Исключение `RuntimeError` перехватывается `rescue`-блоком, а исключение `IOError` не перехватывается и приводит к генерации сообщения об ошибке.

Допускается создание нескольких `rescue`-блоков — под разные типы исключений (листинг 26.12).

Листинг 26.12. Несколько `rescue`-блоков. Файл few_rescue.rb

```
begin
  if rand(2).zero?
    raise RuntimeError, 'Ошибка класса RuntimeError'
  else
    raise IOError, 'Ошибка класса IOError'
  end
rescue RuntimeError => e
  puts e.message
rescue IOError => e
  puts e.message
end
```

Несколько rescue-блоков используются только в том случае, когда содержимое этих блоков различается. В листинге 26.12 они полностью одинаковые, поэтому для обработки обоих типов ошибок можно использовать один блок. Для этого классы исключений приводятся через запятую после ключевого слова rescue (листинг 26.13).

Листинг 26.13. Один rescue-блок. Файл one_rescue.rb

```
begin
  if rand(2).zero?
    raise RuntimeError, 'Ошибка класса RuntimeError'
  else
    raise IOError, 'Ошибка класса IOError'
  end
rescue RuntimeError, IOError => e
  puts e.message
end
```

26.6. Многократная попытка выполнить код

Блок кода, расположенный между ключевыми словами begin и rescue, можно попытаться выполнить несколько раз. Для этого используется ключевое слово retry. В листинге 26.14 перед тем, как завершить работу программы, содержимое begin-блока выполняется три раза.

Листинг 26.14. Многократная попытка выполнить begin-блок. Файл retry.rb

```
tries = 0
begin
  tries += 1
  puts "Попытка #{tries}..."
  raise 'Ошибка'
rescue
  retry if tries < 3
  puts 'Попытки закончились'
end
```

Результатом выполнения программы будут следующие строки:

```
Попытка 1...
Попытка 2...
Попытка 3...
Попытки закончились
```

26.7. Перехват исключений: почти всегда плохо

Не стоит слишком увлекаться перехватом исключительных ситуаций. В 95% случаев лучше завершить работу программы с ошибкой, чем пытаться перехватить ее. Перехват ошибки часто маскирует проблему, транслирует ошибку далеко от места ее возникновения. Вместо того, чтобы быстро обнаружить источник проблем, ошибка расползается по всей системе, приводя к необходимости адаптировать весь остальной код.

«Падать» лучше как можно быстрее, оперативно исправляя проблему по возможности ближе к источнику ее возникновения.

Можно подавлять любые исключения и ошибки — например, при помощи конструкции `rescue nil`, но поступать так не следует — такой код создает больше проблем, чем решает (листинг 26.15).

Листинг 26.15. Никогда так не делайте! Файл rescue_nil.rb

```
arr = []
result = arr[:hello][:world] rescue nil
```

26.8. Блок *ensure*

Механизм исключений, помимо блоков `rescue`, предоставляет `ensure`-блоки, которые выполняются в любом случае, независимо от того, генерируется исключение или нет. Проще всего показать работу такого блока внутри метода (листинг 26.16).

Листинг 26.16. Использование блока `ensure`. Файл ensure.rb

```
def say
  begin
    raise 'Ошибка' if rand(2).zero?
    puts 'Ошибки нет'
  ensure
    return 'Возвращаемое значение'
  end
end

puts say # Возвращаемое значение
```

Здесь внутри метода случайным образом генерируется исключение. Если ошибка не возникает, выводится сообщение `'Ошибки нет'`. При этом, независимо от того, генерируется исключение или нет, обязательно срабатывает содержимое `ensure`-блока.

Внутри методов можно не использовать ключевые слова `begin` и `end` — роль `begin` может играть ключевое слово `def` (листинг 26.17).

Листинг 26.17. Сокращенная форма обработчика исключений. Файл def.rb

```
def say
  raise 'Ошибка' if rand(2).zero?
rescue
  puts 'Только в случае ошибки'
ensure
  puts 'В любом случае'
end

say
```

В зависимости от того, генерируется исключение или нет, программа будет выводить, либо:

```
Только в случае ошибки
В любом случае
```

либо:

```
В любом случае
```

26.9. Блок *else*

Исключения предоставляют механизм для выполнения кода в случае исключения — `rescue`-блок. При помощи `ensure`-блока обеспечивается механизм выполнения кода в любом случае — было исключение или нет. Однако исключения предоставляют еще один блок — `else`, который выполняет код только в том случае, если исключения не произошло.

В листинге 26.18 приводится модифицированный пример метода `say`, в котором используется блок `else`.

Листинг 26.18. Использование блока `else`. Файл def_else.rb

```
def say
  raise 'Ошибка' if rand(2).zero?
rescue
  puts 'Только в случае ошибки'
else
  puts 'Только в случае успешного выполнения'
ensure
  puts 'В любом случае'
end

say
```

В зависимости от того, генерируется исключение или нет, программа будет выводить, либо:

```
Только в случае ошибки
В любом случае
```

либо:

```
Только в случае успешного выполнения
В любом случае
```

26.10. Перехват исключений в блоке

Начиная с версии Ruby 2.6, допускается использование ключевых слов `rescue`, `else` и `ensure` внутри блоков. В листинге 26.19 приводится пример блока, в котором перехватывается исключение попытки деления на ноль `ZeroDivisionError`.

Листинг 26.19. Исключения в блоках. Файл blocks.rb

```
arr = [1, 2, 0, 4].map do |number|
        10 / number
      rescue ZeroDivisionError => e
        puts "Возникла ошибка: #{e.message}"
      end

p arr
```

Результатом выполнения программы будут следующие строки:

```
Возникла ошибка: divided by 0
[10, 5, nil, 2]
```

Задания

1. Создайте класс пользователя `User`, объект которого может хранить фамилию, имя и отчество пользователя, а также его электронный адрес. Кроме того, создайте исключение `UserException`, которое выбрасывается при попытке ввести неверный электронный адрес или имя пользователя, содержащее символы, отличные от русского языка.
2. Создайте класс Солнечной системы, который предоставляет методы, совпадающие с названиями планет. При вызове эти методы должны сообщать порядковый номер планеты, считая от Солнца. При попытке обратиться к несуществующей планете, класс должен выбрасывать исключение `NotExistingPlanetException`.

ГЛАВА 27

Файлы

Файлы с исходными кодами этой главы находятся в каталоге *files* сопровождающего книгу электронного архива.

Файлы — это именованные участки на жестком диске, предназначенные для долговременного хранения информации. Они располагаются в древовидных каталогах, которые могут быть вложены друг в друга.

Концепция файловой системы разрабатывалась во времена, когда объектно-ориентированных языков программирования не существовало. Тем не менее Ruby представляет файлы и каталоги как объекты.

27.1. Класс *IO*

Объекты класса `IO` представляют потоки ввода и вывода в Ruby. Потоки используются для чтения информации с клавиатуры, вывода информации на монитор компьютера, чтения и записи на жесткий диск.

Мы уже неявно использовали их, выводя информацию в консоль при помощи методов `p` и `puts` и получая ее от пользователей при помощи метода `gets`.

Системные вызовы для работы с подсистемой ввода/вывода, как правило, оформлены в виде библиотечных Си-процедур. Язык Си довольно-таки старый и не поддерживает объектно-ориентированное программирование. Ruby-класс `IO` и его производные классы предоставляют объектно-ориентированный интерфейс для взаимодействия с потоками и файлами.

В этой главе мы научимся создавать свои собственные потоки ввода/вывода. Однако любая Ruby-программа по умолчанию снабжается тремя стандартными потоками: ввода, вывода и потока ошибок. Для этих трех потоков Ruby предоставляет три предопределенные константы:

- `STDIN` — стандартный поток ввода, связанный по умолчанию с клавиатурой. Все, что пользователь вводит, попадает в стандартный поток ввода;
- `STDOUT` — стандартный поток вывода, связанный по умолчанию с консолью;

- STDERR — поток ошибок, в который информация по умолчанию не выводится. Получить к этому потоку доступ можно только явно обратившись к константе STDERR.

Обратившись к любой из указанных констант, можно убедиться, что они ссылаются на объект класса IO:

```
 > STDIN.class
=> IO
 > STDOUT.class
=> IO
 > STDERR.class
=> IO
```

Обычно мы используем потоки ввода/вывода неявно, однако константы STDIN, STDOUT и STDERR можно использовать в качестве получателей методов (листинг 27.1).

Листинг 27.1. Явное использование стандартных потоков. Файл io.rb

```
puts 'Hello, world!'        # Hello, world!
STDOUT.puts 'Hello, world!' # Hello, world!
STDERR.puts 'Ошибка'        # Ошибка
```

Помимо предопределенных констант, Ruby предоставляет для стандартных потоков глобальные переменные:

```
 > $stdout.class
=> IO
 > $stdin.class
=> IO
 > $stderr.class
=> IO
```

Как правило, к ним прибегают в том случае, когда стандартный поток необходимо переопределить, — например, направив его в файл.

27.2. Создание файла

Для работы с файлами в Ruby предназначен класс File, который наследуется от класса IO:

```
 > File.ancestors
=> [File, IO, File::Constants, Enumerable, Object, Kernel, BasicObject]
```

Получить объект класса File можно, обратившись к его методу new, которому первым аргументом передается название файла, а вторым — режим его открытия (листинг 27.2).

Листинг 27.2. Явное использование стандартных потоков. Файл file.rb

```
file = File.new('hello.txt', 'w')
p file #<File:hello.txt>
```

В результате работы программы будет создан пустой файл hello.txt. Для того чтобы записать информацию в файл, можно воспользоваться уже знакомым нам методом `puts`. Только в качестве получателя будет выступать не стандартный поток, а объект файла (листинг 27.3).

Листинг 27.3. Запись в файл. Файл puts.rb

```
file = File.new('hello.txt', 'w')
file.puts 'Цена билета: 500'
file.puts 'Дата: 2019.10.10 20:20'
file.puts 'Место: 3 ряд, 10 место'
```

В результате выполнения этой программы в файл hello.txt будут записаны следующие строки:

```
Цена билета: 500
Дата: 2019.10.10 20:20
Место: 3 ряд, 10 место
```

Объект класса `File` содержит не всю необходимую для работы с файлом информацию. Так как файлом управляет операционная система, часть объекта представлена в виде структуры на уровне ядра — ее называют *дескриптором* открытого файла. В рамках одной программы каждый дескриптор имеет уникальный номер, узнать который можно при помощи метода `fileno` (листинг 27.4).

Листинг 27.4. Получение дескриптора открытого файла. Файл fileno.rb

```
file = File.new('hello.txt', 'w')
puts file.fileno # 10
```

Стандартные потоки тоже являются открытыми файлами, а поскольку они открываются первыми, идентификаторы их дескрипторов начинаются с нуля:

```
 > STDIN.fileno
=> 0
 > STDOUT.fileno
=> 1
 > STDERR.fileno
=> 2
```

27.3. Режимы открытия файла

Второй аргумент метода `new` задает режим открытия файла. Все возможные режимы собраны в табл. 27.1.

Таблица 27.1. *Режимы открытия файла*

Режим	Чтение	Запись	Файловый указатель	Очистка файла	Создать, если файла нет
`r`	Да	Нет	В начале	Нет	Нет
`r+`	Да	Да	В начале	Нет	Нет
`w`	Нет	Да	В начале	Да	Да
`w+`	Да	Да	В начале	Да	Да
`a`	Нет	Да	В конце	Нет	Да
`a+`	Да	Да	В конце	Нет	Да

Буквы `r`, `w` и `a` являются сокращениями для следующих английских слов:

- read — чтение;
- write — запись;
- append — добавление.

Как видно из табл. 27.1, для создания файлов подходят любые режимы, кроме `r` и `r+`. При попытке открыть несуществующий файл в одном из этих режимов метод `new` возвращает ошибку (листинг 27.5).

Листинг 27.5. Попытка открытия несуществующего файла. Файл none_exists.rb

```
file = File.new('none_exists.txt', 'r')
```

Эта программа завершится ошибкой, которая сообщает, что файл `none_exists.txt` не существует:

```
No such file or directory @ rb_sysopen - none_exists.txt (Errno::ENOENT)
```

Второй аргумент метода `new` необязательный — если он не указывается, считается, что файл открывается в режиме чтения `r`.

Файл можно представить как последовательность байтов, по которой перемещается файловый указатель (рис. 27.1). Чтение и запись производятся с текущего положения файлового указателя от начала к концу. Движение файлового указателя в обратном порядке не допускается.

При открытии файла в режимах `r` и `w` файловый указатель устанавливается в начало. Это позволяет читать содержимое с начала файла, однако запись в файл также

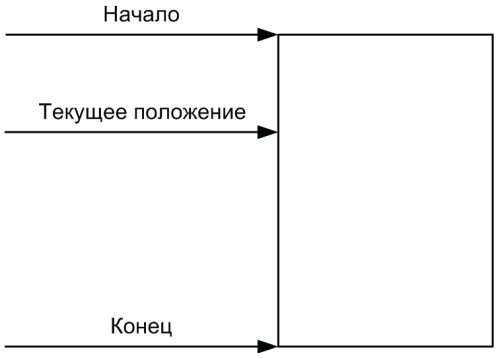


Рис. 27.1. Последовательное перемещение файлового указателя по файлу

производится с начальной позиции, и все старое содержимое файла перезаписывается.

Для того чтобы записать файл, не стирая старые данные, файл следует открыть в режиме `a`, при котором файловый указатель помещается в конец файла. Несмотря на то, что режим `a+` допускает чтение, сразу после открытия файла прочитать ничего не удастся, т. к. файловый указатель находится в конце.

В листинге 27.6 приводится пример добавления фразы `'Hello, world!'` в конец файла hello.txt. Ранее в этот файл уже были записаны три строки при помощи программы из листинга 27.3. Использование режима `a` не позволит затереть эту информацию.

Листинг 27.6. Добавление строки в конец файла. Файл append.rb

```
file = File.new('hello.txt', 'a')
file.puts 'Hello, world!'
```

В результате выполнения этой программы файл hello.txt будет содержать следующие строки:

```
Цена билета: 500
Дата: 2019.10.10 20:20
Место: 3 ряд, 10 место
Hello, world!
```

Помимо представленных в табл. 27.1 режимов, Ruby поддерживает текстовый и бинарный режимы открытия файлов.

Переводы строк в Windows кодируются последовательностью: `\r\n`, в UNIX-подобных операционных системах для перевода строки используется один символ: `\n`. При открытии файла в *текстовом режиме* переводы строк автоматически преобразуются в зависимости от текущей операционной системы. Для того чтобы открыть файл в таком режиме, во второй параметр метода `new` следует добавить символ `t` (листинг 27.7).

Листинг 27.7. Открытие файла в текстовом режиме. Файл text_mode.rb

```
file = File.new('hello.txt', 'a+t')
file = File.new('hello.txt', 'rt')
file = File.new('hello.txt', 'wt')
```

По умолчанию файл открывается в *бинарном режиме*, в котором никаких манипуляций с переводами строк не проводится. Чтобы явно сообщить об открытии файла в бинарном режиме, во второй аргумент можно поместить символ `b` (листинг 27.8). Впрочем, на практике явное указание бинарного режима часто опускается.

Листинг 27.8. Открытие файла в бинарном режиме. Файл binary_mode.rb

```
file = File.new('hello.txt', 'a+b')
file = File.new('hello.txt', 'rb')
file = File.new('hello.txt', 'wb')
```

27.4. Закрытие файла

После выполнения всех необходимых действий с файлом его можно закрыть с помощью метода `close`. При этом дескриптор файла уничтожается, и следующий открытый файл будет использовать номер закрытого файла (листинг 28.9).

Листинг 27.9. Явное закрытие файла. Файл close.rb

```
file = File.new('hello.txt', 'w')
puts file.fileno # 10
file.puts 'Цена билета: 500'
file.puts 'Дата: 2019.10.10 20:20'
file.puts 'Место: 3 ряд, 10 место'
file.close

other = File.new('another.txt', 'w')
puts other.fileno # 10
other.close
```

Закрывать файл при помощи метода `close` необязательно, т. к. все открытые файлы закрываются автоматически после завершения работы программы. Однако лучше все-таки использовать метод `close` — особенно, если в файл осуществлялась запись информации. Дело в том, что информация записывается в файл не побайтово, а блоками — по мере заполнения буфера записи. Содержимое буфера сбрасывается на диск либо при его заполнении, либо при использовании метода `flush`, либо при закрытии файла методом `close`. Если работа скрипта будет неожиданно остановлена, информация из буфера, не сброшенная на жесткий диск, пропадет.

Кроме того, открытые файлы — это исчерпаемый ресурс: одновременно можно открыть лишь ограниченное количество файлов. Это значение обычно велико, но конечно. Поэтому открытые файлы, которые больше не нужны для работы, следует закрывать.

Проверить, открыт или закрыт файл, можно при помощи логического метода `closed?`. Метод возвращает `true`, если файл закрыт, иначе возвращается `false` (листинг 27.10).

Листинг 27.10. Проверка, открыт или закрыт файл. Файл closed.rb

```
file = File.new('hello.txt')
p file.closed? # false
file.close
p file.closed? # true
```

Для `new` имеется метод-синоним `open`, который ведет себя точно так же (листинг 27.11).

Листинг 27.11. Использование метода open. Файл open.rb

```
file = File.open('hello.txt', 'w')
file.puts 'Цена билета: 500'
file.puts 'Дата: 2019.10.10 20:20'
file.puts 'Место: 3 ряд, 10 место'
file.close
```

В результате работы этой программы в файл hello.txt будут записаны следующие строки:

```
Цена билета: 500
Дата: 2019.10.10 20:20
Место: 3 ряд, 10 место
```

В отличие от `new`, метод `open` может принимать блок с единственным параметром — объектом открытого файла. Открытый файл автоматически закрывается при выходе из блока (листинг 27.12).

Листинг 27.12. Использование блока совместно с методом open. Файл open_block.rb

```
File.open('hello.txt', 'w') do |file|
  file.puts 'Цена билета: 500'
  file.puts 'Дата: 2019.10.10 20:20'
  file.puts 'Место: 3 ряд, 10 место'
end
```

27.5. Чтение содержимого файла

Прочитать содержимое файла можно при помощи метода `read`. В листинге 27.13 извлекается все содержимое файла hello.txt.

Листинг 27.13. Чтение файла. Файл read.rb

```
File.open('hello.txt') do |file|
  p file.read
end
```

В результате выполнения этой программы содержимое файла будет выведено в виде строки:

```
"Цена билета: 500\nДата: 2019.10.10 20:20\nМесто: 3 ряд, 10 место\n"
```

При этом отдельные строки файла отделяются друг от друга переводом строки: `\n`. Впрочем, прочитать содержимое файла можно более простым способом, воспользовавшись методом `File.read` (листинг 27.14).

Листинг 27.14. Чтение содержимого файла. Файл file_read.rb

```
p File.read('hello.txt')
```

27.6. Построчное чтение файла

Для построчного чтения содержимого файла можно воспользоваться методом `gets` (листинг 27.15).

Листинг 27.15. Построчное чтение файла. Файл gets.rb

```
File.open('hello.txt') do |file|
  puts file.gets # Цена билета: 500
  puts file.gets # Дата: 2019.10.10 20:20
  puts file.gets # Место: 3 ряд, 10 место
  p file.gets    # nil
end
```

По достижении конца файла метод `gets` возвращает неопределенное значение `nil`. Для чтения строк файла вместо метода `gets` можно использовать метод `readline`. При попытке чтения строк, когда конец файла уже достигнут, этот метод генерирует исключение (листинг 27.16).

Листинг 27.16. Построчное чтение файла. Файл readline.rb

```
File.open('hello.txt') do |file|
  puts file.readline # Цена билета: 500
  puts file.readline # Дата: 2019.10.10 20:20
```

```
  puts file.readline # Место: 3 ряд, 10 место
  puts file.readline # end of file reached (EOFError)
end
```

Для определения этого состояния предусмотрен специальный логический метод `eof?`. Он возвращает `true`, если достигнут конец файла, и `false`, если это не так. Метод `eof?` специально предназначен для совместной работы с циклами `while` или `until` (листинг 27.17).

Листинг 27.17. Использование метода `eof?`. Файл eof.rb

```
File.open('hello.txt') do |file|
  until file.eof?
    puts file.gets
  end
end
```

Впрочем, можно обойтись без метода `eof?` — для этого метод `gets` следует поместить в условие цикла (листинг 27.18).

Листинг 27.18. Использование метода `gets` в условии цикла. Файл while.rb

```
File.open('hello.txt') do |file|
  while line = file.gets
    puts line
  end
end
```

Операция присваивания `line = file.gets` каждый раз возвращает строку. Так как это значение отлично от `false` и `nil`, тело цикла выполняется до тех пор, пока не будет достигнут конец файла. Попытка чтения файла методом `gets` в этом случае вернет `nil`, что в контексте цикла `while` станет рассматриваться как ложное значение. В результате цикл автоматически завершит работу.

В цепочке поиска метода класса `File` присутствует модуль `Enumerable` (см. *разд. 21.4*):

```
 > File.ancestors
=> [File, IO, File::Constants, Enumerable, Object, Kernel, BasicObject]
```

Таким образом, с содержимым файла можно обращаться как с коллекцией. Поэтому для построчного чтения файла чаще прибегают к итератору `each` (листинг 27.19).

Листинг 27.19. Построчное чтение файла итератором `each`. Файл each.rb

```
File.open('hello.txt') do |file|
  file.each { |line| puts line }
end
```

Результатом работы этой программы будет построчный вывод содержимого файла:

```
Цена билета: 500
Дата: 2019.10.10 20:20
Место: 3 ряд, 10 место
```

При помощи итератора `each_with_index` можно пронумеровать строки (листинг 27.20).

Листинг 27.20. Использование итератора `each_with_index`. Файл each_with_index.rb

```
File.open('hello.txt') do |file|
  file.each_with_index { |line, i| puts "#{i + 1}. #{line}" }
end
```

Результатом работы этой программы будут пронумерованные строки файла:

```
1. Цена билета: 500
2. Дата: 2019.10.10 20:20
3. Место: 3 ряд, 10 место
```

Получить содержимое файла в виде массива строк можно при помощи итератора `map`. Для того чтобы сократить блок `{ |line| line }`, можно задействовать метод `itself`, который возвращает объект (листинг 27.21).

> ***Замечание***
>
> Метод `itself` был добавлен в класс `Object`, начиная с Ruby версии 2.2.

Листинг 27.21. Получение содержимого файла в виде массива строк. Файл map.rb

```
arr = File.open('hello.txt') do |file|
        file.map(&:itself)
      end

p arr
```

Результатом работы этой программы будет следующий массив:

```
[
  "Цена билета: 500\n",
  "Дата: 2019.10.10 20:20\n",
  "Место: 3 ряд, 10 место\n"
]
```

Получить содержимое файла в виде массива строк можно и более простым способом, если воспользоваться методом `File.readlines` (листинг 27.22).

Листинг 27.22. Использование метода `File.readlines`. Файл readlines.rb

```
arr = File.readlines('hello.txt')
p arr
```

Элементы полученного массива включают переводы строк. От них легко избавиться при помощи метода `chomp`, который можно применить к каждому элементу массива при помощи итератора `map` (листинг 27.23).

Листинг 27.23. Очищаем строки файла от \n. Файл chomp.rb

```
arr = File.readlines('hello.txt').map(&:chomp)
p arr
```

Результатом выполнения этой программы будет следующий массив:

```
[
  "Цена билета: 500",
  "Дата: 2019.10.10 20:20",
  "Место: 3 ряд, 10 место"
]
```

Впрочем, метод `readlines` принимает дополнительный параметр `chomp`, при помощи которого можно потребовать очищать строки от последовательностей `\r\n` и `\n` (листинг 27.24).

Листинг 27.24. Файл readlines_chomp.rb

```
arr = File.readlines('hello.txt', chomp: true)
p arr
```

27.7. Запись в файл

Для того чтобы записать информацию в файл, он должен быть открыт в режиме записи (см. *разд. 27.3*). Для этого методу `new` или `open` в качестве второго аргумента необходимо передать строку `'w'` или `'w+'` (листинг 27.25).

Листинг 27.25. Запись информации в файл. Файл file_puts.rb

```
File.open('hello.txt', 'w') do |file|
  file.puts 'Цена билета: 500'
  file.puts 'Дата: 2019.10.10 20:20'
  file.puts 'Место: 3 ряд, 10 место'
end
```

Метод `puts` автоматически добавляет в конец перевод строки. В том случае, если этого не требуется, можно воспользоваться методом `write` (листинг 27.26).

Листинг 27.26. Запись в файл при помощи метода write. Файл write.rb

```
File.open('hello.txt', 'w') do |file|
  file.puts 'Hello'
```

```
  file.puts ' '
  file.puts 'world!'

  file.write 'Hello'
  file.write ' '
  file.write 'world!'
end
```

В результате в файл hello.txt будут записаны четыре строки:

```
Hello

world!
Hello world!
```

Первые три строки сформированы методом `puts`, последняя — тремя вызовами метода `write`.

Оба метода: `puts` и `write` — могут принимать несколько аргументов. Предыдущую программу можно переписать короче (листинг 27.27).

Листинг 27.27. Запись сразу нескольких аргументов. Файл write_few_args.rb

```
File.open('hello.txt', 'w') do |file|
  file.puts 'Hello', ' ', 'world!'
  file.write 'Hello', ' ', 'world!'
end
```

Прием нескольких аргументов методами `puts` и `write` позволяет записывать в файл массивы. Для этого массив можно разложить на отдельные элементы при помощи оператора `*` (листинг 27.28).

Листинг 27.28. Запись массива в файл. Файл splat.rb

```
lines = ['Hello', ' ', 'world!']

File.open('hello.txt', 'w') do |file|
  file.puts *lines
end
```

Впрочем, эффективнее объединить элементы массива при помощи метода `join` и записать полученную строку в файл (листинг 27.29).

Листинг 27.29. Использование метода `join`. Файл join.rb

```
lines = ['Hello', ' ', 'world!']

File.open('hello.txt', 'w') do |file|
  file.write lines.join
end
```

Для записи в файл одной строки можно не открывать файл, а воспользоваться методом `File.write` (листинг 27.30).

Листинг 27.30. Запись в файл при помощи метода `File.write`. Файл file_write.rb

```
File.write('hello.txt', 'Hello, world!')
```

Метод класса `write` принимает в качестве первого аргумента имя файла, а в качестве второго — записываемую строку.

27.8. Произвольный доступ к файлу

Запись или чтение внутри файла начинается с точки, определяемой файловым указателем (см. рис. 27.1). При открытии файла он может быть установлен в его начало или в конец. При чтении или записи какого-то объема информации файловый указатель автоматически смещается на соответствующее количество байтов.

Все рассмотренные до этого методы оперировали файловым указателем неявно. Однако в управление файловым указателем допускается вмешиваться. Так, определять текущее положение файлового указателя можно при помощи метода `pos`.

В листинге 27.31 открывается файл hello.txt, в который последовательно записываются несколько строк. После каждой операции методом `pos` запрашивается текущее положение файлового указателя. Метод возвращает количество байтов от начала файла.

> ***Замечание***
>
> Метод `pos` имеет синоним `tell`.

Листинг 27.31. Определение положения файлового указателя. Файл pos.rb

```
File.open('hello.txt', 'w') do |file|
  puts file.pos                      # 0
  file.puts 'Цена билета: 500'
  puts file.pos                      # 27
  file.puts 'Дата: 2019.10.10 20:20'
  puts file.pos                      # 54
  file.puts 'Место: 3 ряд, 10 место'
  puts file.pos                      # 90
end
```

Следует учитывать, что в кодировке UTF-8 под разные символы отводится различное количество байтов. Под русские символы — два байта. Под пробел, двоеточие и цифры — один байт. Кроме того, перевод строки (`\n`) в UNIX-подобных операционных системах занимает один байт, а в Window (`\r\n`) — два байта (рис. 27.2).

Часто при чтении или записи файловый указатель оказывается в конце файла, и для того, чтобы осуществить повторное чтение, требуется переместить его в начало.

Этого эффекта можно добиться, закрыв файл и открыв его по-новой. Однако проще всего возвратить файловый указатель в начало файла при помощи метода `rewind`.

В листинге 27.32 программа выводит содержимое файла, переданного ей в качестве аргумента. Перед содержимым файла выводится количество строк в файле. Для получения количества строк содержимое файла читается построчно при помощи итератора `each`. На каждой итерации значение счетчика `count` увеличивается на единицу.

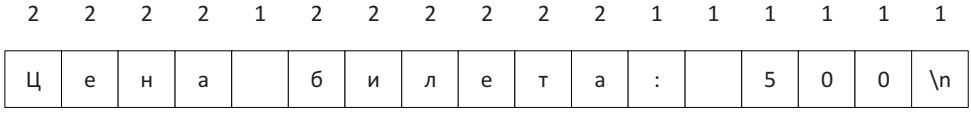


Рис. 27.2. В UTF-8 русские буквы занимают два байта, цифры, знаки препинания, пробельные символы — один

Листинг 27.32. Перенос файлового указателя в начало файла. Файл rewind.rb

```
count = 0

File.open(ARGV.first, 'r') do |f|
  f.each { |_line| count += 1 }

  puts "Строк в файле: #{count}"

  f.rewind
  f.each { |line| puts line }
end
```

Программа ожидает в качестве аргумента имя файла, который извлекается из массива параметров `ARGV` (см. *главу 6*).

```
$ ruby rewind.rb rewind.rb
Строк в файле: 10
count = 0
...
```

После подсчета количества строк файловый указатель оказывается в конце файла. Поэтому, чтобы вывести содержимое файла, его необходимо снова переместить в начало, для чего используется метод `rewind` (рис. 27.3). После этого можно опять выводить строки при помощи метода `each`.

Для установки файлового указателя в произвольную позицию можно воспользоваться методом `seek`.

В UNIX-подобных операционных системах популярна команда `tail`, которая позволяет извлечь из текстового файла несколько последних строк. Для демонстрации метода `pos=` разработаем урезанную версию этой программы, которая выводит из текстового файла три последние строки.

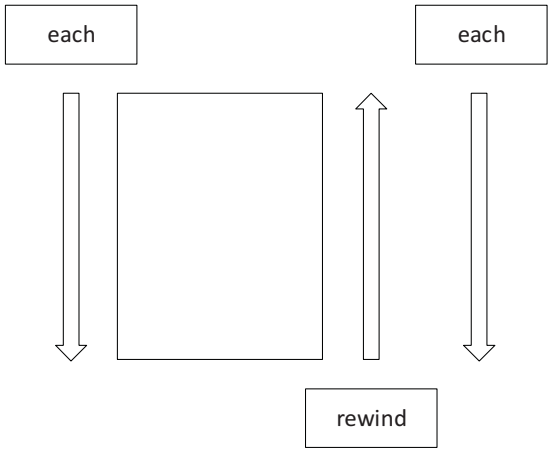


Рис. 27.3. Метод `rewind` позволяет вернуть файловый указатель в начало файла

Можно прочитать содержимое файла, сохраняя в массиве последние три строки файла, однако экономнее хранить не сами строки, а начальные их позиции. Для этого следует завести массив `positions` из четырех элементов и на каждой итерации помещать в его начало текущую позицию файлового указателя, а с конца массива убирать один элемент.

В листинге 27.33 приводится возможная реализация заполнения массива начальными позициями строк файла.

Листинг 27.33. Извлечение начальных позиций строк файла. Файл positions.rb

```
positions = Array.new(4)

File.open(ARGV.first, 'r') do |f|
  f.map do |line|
    positions.pop
    positions.unshift f.pos
  end
end

p positions # [147, 135, 134, 130]
```

Результатом запуска этой программы будет массив, в котором первый элемент соответствует концу файла, а второй, третий и четвертый — начальным позициям последних строк:

```
$ ruby positions.rb positions.rb
[147, 135, 134, 130]
```

Теперь остается лишь переместить файловый указатель в позицию, которая соответствует третьей строке с конца, и повторно выполнить чтение файла.

В листинге 27.34 приводится окончательная реализация программы.

Листинг 27.34. Использование метода seek. Файл tail.rb

```
positions = Array.new(4)

File.open(ARGV.first, 'r') do |f|
  f.each do |line|
    positions.pop
    positions.unshift f.pos
  end

  f.seek positions.last

  f.each do |line|
    puts line
  end
end
```

Результатом выполнения этой программы будут последние три строчки файла:

```
$ ruby tail.rb tail.rb
    puts line
  end
end
```

27.9. Пути к файлам

Программа может оперировать множеством файлов, и чтобы их как-то отличать друг от друга, используется либо имя файла, либо его дескриптор. Внутри каталога разные файлы должны иметь разные имена, в разных каталогах могут размещаться файлы с одинаковыми именами. При этом каталоги можно вкладывать друг в друга. Полное имя файла определяется последовательностью каталогов от корня диска и именем самого файла (рис. 27.4). Такая последовательность каталогов и названий файлов называется *путем к файлу*.

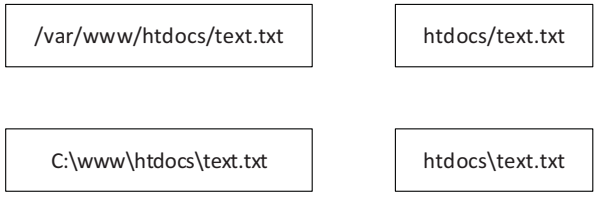


Рис. 27.4. Пути к файлам

В различных операционных системах между каталогами и файлами используются разные разделители. В UNIX-подобных операционных системах в качестве разделителя применяется прямой слеш / (рис. 27.4, *вверху*), в Windows — обратный \ (рис. 27.4, *внизу*). Обратный слеш традиционно служит для экранирования в стро-

ках (см. *разд. 2.4.2*), поэтому его использование при работе в Windows доставляет массу неудобств.

Замечание

В операционных системах Windows разделы обозначаются символами английского алфавита. Так, корень диска обозначается одной из букв — например, `C:\`. В UNIX-подобных операционных системах любой путь начинается с корневого раздела (root-раздела) `/`, к каталогам которого монтируются физические разделы. Таким образом, если в Windows может быть несколько корневых точек, обозначенных буквами, в UNIX-подобных операционных системах такая точка всегда одна.

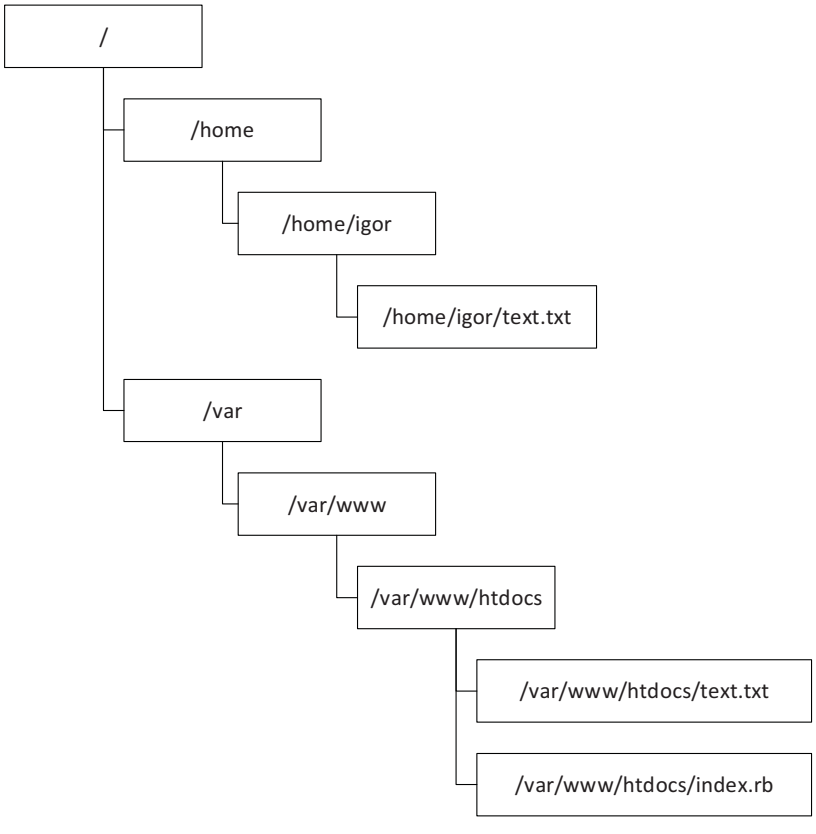


Рис. 27.5. Пример структуры файловой системы

На рис. 27.5 представлен пример структуры UNIX-подобной файловой операционной системы.

Путь от корня диска называется *абсолютным*:

- `/var/home/htdocs/text.txt` — в UNIX-подобной файловой операционной системе;
- `C:\www\htdocs\text.txt` — в Windows.

Относительный путь выстраивается относительно текущего каталога. Как правило, это каталог, в котором расположен Ruby-файл. Так, если текущим каталогом явля-

ется каталог C:\www\htdocs, то для доступа к нижележащим каталогам достаточно указать путь относительно текущего каталога: htdocs\text.txt.

Следует отметить, что относительный путь не предваряет ни буква раздела, ни символ корневого раздела / — он начинается либо с имени каталога, либо с имени файла.

Для того чтобы использовать относительный путь для доступа к файлу, находящемуся в ином каталоге, — например, в C:\dir\text.txt, необходимо подняться на один уровень выше и спуститься в каталог dir. Для формирования таких путей используется последовательность из двух точек .., обозначающая родительский каталог (листинг 27.35).

> ***Замечание***
>
> Помимо родительского каталога, обозначаемого двумя точками (..), большинство файловых систем поддерживает текущий каталог, который обозначается одной точкой.

Листинг 27.35. Доступ к родительскому каталогу. Файл parent_dir.rb

```
file = File.new('../hello.txt', 'w')
p file #<File:hello.txt>
```

Если необходимо подняться еще выше, можно использовать несколько последовательностей .., например: ../../../result/text.txt.

При формировании путей к файлам довольно легко ошибиться, добавив лишний слеш:

```
 > '/home/i.simdyanov/' + '/code/file_join.rb'
=> "/home/i.simdyanov//code/file_join.rb"
```

Избежать этого можно при помощи метода join класса File. Метод принимает в качестве параметров произвольное количество фрагментов пути и формирует конечный путь (листинг 27.36).

Листинг 27.36. Формирование пути при помощи метода File.join. Файл file_join.rb

```
path = '/home/igor/'
p File.join(path, '/code/file_join.rb') # "/home/igor/code/file_join.rb"
p File.join('root', 'etc') # "root/etc"
p File.join('/', 'root', 'etc') # "/root/etc"
```

27.10. Манипуляция файлами

Можно не только изменять содержимое файла, но и оперировать файлом как единым целым. Допускается переименование, удаление файла, а также получение его размера.

Все эти операции не требуют открытия файла и реализованы как синглетон-методы класса `File`:

- `rename` — переименование файла;
- `unlink` — удаление файла;
- `size` — получение размера файла.

Метод `rename` принимает в качестве первого параметра имя существующего файла, а в качестве второго — новое имя файла. В листинге 27.37 файл hello.txt переименовывается в new_hello.txt.

Листинг 27.37. Переименование файла. Файл rename.rb

```
File.rename('hello.txt', 'new_hello.txt')
```

При помощи метода `rename` можно не только переименовывать файлы, но и перемещать их. Для этого вместо имени файла необходимо указать относительный или абсолютный путь. В листинге 27.38 файл another.txt перемещается на один уровень выше.

Листинг 27.38. Перемещение файла. Файл replace.rb

```
File.rename('another.txt', '../another.txt')
```

Метод `unlink` принимает единственный параметр — имя файла. При успешном выполнении метода файл удаляется (листинг 27.39).

> ***Замечание***
> Метод `unlink` имеет синоним `delete`.

Листинг 27.39. Удаление файла. Файл unlink.rb

```
File.unlink('hello.txt')
```

Получить размер файла в байтах можно при помощи метода `size`. Метод принимает в качестве параметра единственный аргумент — путь к файлу (листинг 27.40).

Листинг 27.40. Получение размера файла. Файл size.rb

```
puts File.size('hello.txt') # 90
```

Задания

1. Создайте программу, которая принимает в качестве аргумента имя файла и выводит из файла случайную строку.
2. Создайте программу, которая перемешивает строки внутри файла в случайном порядке.

3. Создайте программу, которая принимает в качестве аргумента целое число и имя файла. Если в текущем каталоге файла с таким именем не существует, программа должна создать файл с этим именем. Размер файла при этом должен в точности соответствовать заданному числу, содержимое файла — на ваше усмотрение.
4. Создайте программу, которая выводит самую длинную и самую короткую строки файлов. Кроме того, необходимо вывести количество символов в этих строках.
5. Разработайте программу, которая принимает в качестве аргумента имя файла и возвращает его размер в байтах, килобайтах, мегабайтах или гигабайтах. Если размер файла больше 1024 байтов, рядом с цифрой следует указывать единицу измерения при помощи символов K, M или G.
6. Создайте программу, которая разбивает файл на десять частей. Рядом с существующим файлом должны появиться десять новых файлов, к расширению которых добавляются суффиксы `x01`, `x02`, ..., `x10`. В случае, если размер файла меньше 10 байтов, программа должна сообщить о невозможности произвести разбиение.
7. Независимо от того, как создано изображение формата JPEG, оно содержит в себе дату создания. Создайте скрипт, который автоматически извлекал бы эту дату из содержимого JPEG-файла.

ГЛАВА 28

Права доступа и атрибуты файлов

Файлы с исходными кодами этой главы находятся в каталоге *files_attributes* сопровождающего книгу электронного архива.

Помимо имени и размера, файлы обладают дополнительными атрибутами: временем создания и обновления, правами доступа, типом файла. В этой главе мы узнаем, как можно извлекать и изменять атрибуты.

Многие методы, которые мы здесь рассмотрим, предназначены для UNIX-подобных операционных систем. Они могут вызываться и в Windows, но не возвращают осмысленного результата.

28.1. Типы файлов

Среди объектов файловой системы имеются несколько типов, которые различаются по назначению и приемам работы с ними:

- *обычные файлы* — последовательность байтов, которая хранится на каком-либо физическом носителе (см. *главу 27*);
- *каталоги* — файлы специального типа, в которых содержится список файлов и других каталогов (см. *главу 29*). Обрабатывать каталоги при помощи файловых методов не получится — за этим следит операционная система. Для работы с каталогами предназначен специальный класс `Dir`;
- *жесткие ссылки* — дополнительные точки доступа к файлу. Внутри операционной системы файлы различаются друг от друга по уникальным идентификаторам. Связь названия файла с таким идентификатором и называется *жесткой ссылкой*. Обычно на один файл указывает одна жесткая ссылка, однако многие файловые системы допускают создание на один файл нескольких таких ссылок — тогда к этому файлу можно получить доступ из разных точек файловой системы. Файл невозможно удалить из системы до тех пор, пока не будет удалена последняя указывающая на него жесткая ссылка;
- *символические ссылки* — файлы специального типа, указывающие на другой файл. Внешне они выглядят точно так же, как жесткие ссылки. Однако при уда-

лении оригинального файла такая ссылка в системе останется и будет указывать на несуществующий файл.

Рассмотрим подробнее описанные типы файлов, используя UNIX-команды. Обычный файл можно создать при помощи команды `touch`, которой передается имя файла:

```
$ touch hello.txt
```

> ***Замечание***
>
> В *главе 27* мы уже познакомились со способами создания файлов. Каталоги и способы их создания будут подробно рассмотрены в *главе 29*.

Получить список файлов можно при помощи команды `ls`:

```
$ ls -la
total 0
drwxr-xr-x@  3 i.simdyanov  staff   102 26 янв 17:11 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--   1 i.simdyanov  staff     0 26 янв 17:11 hello.txt
```

> ***Замечание***
>
> В Windows для получения списка файлов текущего каталога используется команда `dir`.

Для создания каталога можно воспользоваться командой `mkdir`, которой в качестве аргумента передается имя каталога:

```
$ mkdir test
$ ls -la
total 0
drwxr-xr-x@  4 i.simdyanov  staff   136 26 янв 17:14 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--@  1 i.simdyanov  staff     0 26 янв 17:11 hello.txt
drwxrwxr-x@  2 i.simdyanov  staff    68 26 янв 17:14 test
```

Жесткая ссылка создается при помощи команды `ln` — первый аргумент команды принимает существующий файл, второй — название жесткой ссылки:

```
$ ln hello.txt newhello.txt
$ ls -la
total 0
drwxr-xr-x@  5 i.simdyanov  staff   170 26 янв 17:14 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--@  2 i.simdyanov  staff     0 26 янв 17:11 hello.txt
-rw-rw-r--@  2 i.simdyanov  staff     0 26 янв 17:11 newhello.txt
drwxrwxr-x@  2 i.simdyanov  staff    68 26 янв 17:14 test
```

Чтобы убедиться, что hello.txt и newhello.txt — это один и тот же файл (т. е. newhello.txt лишь указывает на hello.txt), запишем в файл hello.txt какой-нибудь текст — например, `'Hello, Ruby!'`. Для этого можно воспользоваться командой `echo`:

```
$ echo 'Hello, Ruby!' > hello.txt
```

Извлечь содержимое файла можно при помощи команды `cat`:

```
$ cat hello.txt
Hello, Ruby!
```

Для файла newhello.txt команда `cat` дает тот же самый результат:

```
$ cat newhello.txt
Hello, Ruby!
```

> ***Замечание***
>
> В Windows для получения содержимого файла используется команда `type`.

В файл newhello.txt мы ничего не записывали, но он также содержит строку `'Hello, Ruby!'`. Таким образом получается, что два файла: hello.txt и newhello.txt — это один и тот же файл, просто у него два названия, две точки входа (рис. 28.1).

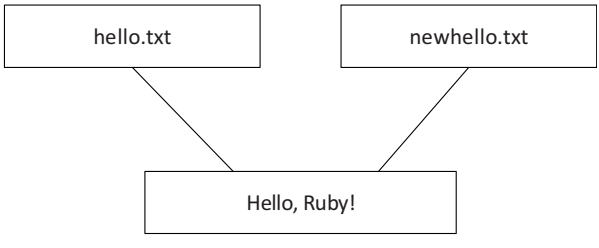


Рис. 28.1. Файлы hello.txt и newhello.txt — один и тот же файл

Удалить файл можно при помощи команды `unlink`. Удалим, например, файл newhello.txt, представляющий собой жесткую ссылку на файл hello.txt :

```
$ unlink newhello.txt
$ ls -la
total 8
drwxr-xr-x@  4 i.simdyanov  staff   136 26 янв 17:18 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--@  1 i.simdyanov  staff    13 26 янв 17:16 hello.txt
drwxrwxr-x@  2 i.simdyanov  staff    68 26 янв 17:14 test
```

При этом файл hello.txt остается без изменений — просто счетчик ссылок на него уменьшился до единицы (рис. 28.2).

Если сейчас удалить файл hello.txt, счетчик ссылок станет равным нулю, и операционная система удалит файл.

> ***Замечание***
>
> Напомню, что операционная система не удаляет файл до тех пор, пока на него ссылается хотя бы одна жесткая ссылка.

Символическая ссылка также создается при помощи команды `ln`, только с параметром `-s`:

```
$ ln -s hello.txt newhello.txt
$ ls -la
```

```
total 16
drwxr-xr-x@  5 i.simdyanov  staff   170 26 янв 17:19 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--@  1 i.simdyanov  staff    13 26 янв 17:16 hello.txt
lrwxrwxr-x   1 i.simdyanov  staff     9 26 янв 17:19 newhello.txt -> hello.txt
drwxrwxr-x@  2 i.simdyanov  staff    68 26 янв 17:14 test
```

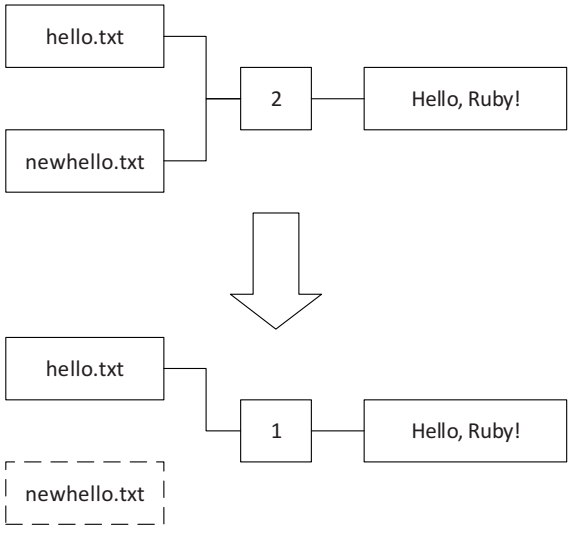


Рис. 28.2. Счетчик файла до удаления файла и после

В отчете команды `ls` символическая ссылка оформляется особым образом. Можно удалить файл, на который ссылается ссылка, — при этом ссылка не исчезает, но манипулировать файлом с ее помощью уже не получится.

Создать символическую ссылку можно при помощи метода `File.symlink`. Метод принимает в качестве первого параметра имя существующего файла, а в качестве второго — название ссылки (листинг 28.1).

Листинг 28.1. Создание символической ссылки. Файл symlink.rb

```
File.symlink('hello.txt', 'newhello.txt')
```

Как уже упоминалось ранее, для создания жесткой ссылки можно использовать метод `ln`. Метод также принимает два параметра: имя файла и название создаваемой ссылки (листинг 28.2).

Листинг 28.2. Создание жесткой ссылки. Файл ln.rb

```
File.ln('hello.txt', 'text.txt')
```

28.2. Определение типа файла

Класс `File` предоставляет логические методы, позволяющие определить тип файла. Так, выяснить, является ли текущий файл обычным файлом, можно методом `file?`, является ли текущий файл каталогом покажет метод `directory?`, а символической ссылкой — метод `symlink?`:

```
 > File.file? 'hello.txt'
=> true
 > File.file? 'test'
=> false
 > File.directory? 'hello.txt'
=> false
 > File.directory? 'test'
=> true
 > File.symlink? 'hello.txt'
=> false
 > File.symlink? 'newhello.txt'
=> true
```

Методы возвращают `true`, если файл существует и соответствует заданному типу, иначе возвращается `false`.

Для проверки существования файла, каталога или ссылки предназначен метод `exist?`:

```
 > File.exist? 'hello.txt'
=> true
 > File.exist? 'none_exist_file.txt'
=> false
```

28.3. Время последнего доступа к файлу

Файловая система предоставляет для каждого файла несколько временных меток, по которым можно отслеживать последние обращения к файлу. Для их получения в классе `File` предусмотрены следующие методы:

- `mtime` — возвращает время последнего изменения содержимого файла;
- `atime` — возвращается время последнего чтения файла;
- `ctime` — возвращает время последнего изменения метаинформации файла (имя, атрибуты, права доступа и т. п.). В Windows метод возвращает время создания файла.

Методы принимают в качестве параметра путь к файлу, а возвращают объект класса `Time`. В листинге 28.3 приводится пример использования всех трех указанных методов.

Листинг 28.3. Временные метки последнего доступа к файлу. Файл time.rb

```
filename = 'time.rb'

puts "Время последнего изменения файла #{File.mtime(filename)}"
puts "Время последнего чтения #{File.atime(filename)}"
puts "Время последнего изменения метаинформации #{File.ctime(filename)}"
```

Результатом выполнения этой программы будут следующие строки:

```
Время последнего изменения файла 2019-01-27 14:16:15 +0300
Время последнего чтения 2019-01-27 14:16:33 +0300
Время последнего изменения метаинформации 2019-01-27 14:16:15 +0300
```

Время последнего изменения и чтения можно установить при помощи метода `utime`. Метод принимает в качестве первого параметра время последнего чтения (`atime`), в качестве второго — время последнего изменения содержимого (`mtime`), а в качестве третьего — путь к файлу (листинг 28.4).

Листинг 28.4. Изменение временны́х меток. Файл utime.rb

```
filename = 'hello.txt'

puts "atime #{File.atime(filename)}" # atime 2019-01-27 14:24:52 +0300
puts "mtime #{File.mtime(filename)}" # mtime 2019-01-26 17:16:15 +0300

time = Time.new(2019, 10, 28, 10, 20)
File.utime(time, time, filename)

puts "atime #{File.atime(filename)}" # atime 2019-10-28 10:20:00 +0300
puts "mtime #{File.mtime(filename)}" # mtime 2019-10-28 10:20:00 +0300
```

В результате выполнения этой программы время доступа к файлу hello.txt устанавливается на 28 октября 2019 года:

```
$ ls -la
drwxr-xr-x@  9 i.simdyanov  staff   306 27 янв 14:23 .
drwxr-xr-x@ 32 i.simdyanov  staff  1088 26 янв 16:55 ..
-rw-rw-r--@  1 i.simdyanov  staff    13 28 окт  2019 hello.txt
drwxrwxr-x@  2 i.simdyanov  staff    68 26 янв 17:14 test
```

Методы `mtime`, `atime` и `ctime` доступны в том числе и как инстанс-методы. Если имеется объект открытого файла, методы можно применить и к нему (листинг 28.5).

Листинг 28.5. Файл time_instance.rb

```
f = File.open('time.rb')

puts "Время последнего изменения файла #{f.mtime}"
puts "Время последнего чтения #{f.atime}"
puts "Время последнего изменения метаинформации #{f.ctime}"
```

28.4. Права доступа в UNIX-подобной системе

Часто можно столкнуться с ситуацией, когда Ruby-программа успешно работает в Windows, однако отказывается работать в UNIX-подобной операционной системе (листинг 28.6).

Листинг 28.6. Попытка чтения файла. Файл permission_denied.rb

```
puts File.read('/etc/sudoers')
```

Все дело тут в установленных правах доступа к читаемому файлу — если прав доступа к нему недостаточно, при выполнении программы может возникнуть ошибка:

```
$ ruby permission_denied.rb
permission_denied.rb:1:in `read': Permission denied @ rb_sysopen - /etc/sudoers
(Errno::EACCES)
```

Права доступа к файлам и каталогам в UNIX-подобных операционных системах задаются для трех категорий пользователей:

- владельца файла;
- группы владельца (все пользователи, входящие в эту группу);
- всех остальных.

Такое разделение пользователей помогает выстраивать права доступа в многопользовательских системах, где множество разных пользователей одновременно обращаются к одним и тем же файлам и каталогам (рис. 28.3).

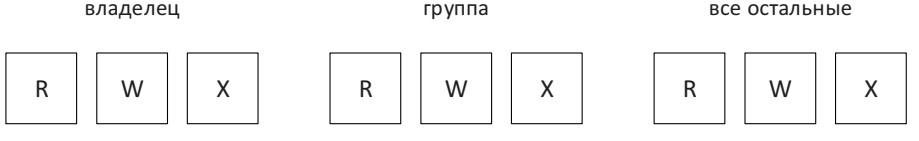


Рис. 28.3. Права доступа в UNIX-подобной операционной системе

В отчете UNIX-команды `ls -la` права доступа сосредоточены в первой позиции:

```
$ ls -la
-rw-rw-r--  1 i.simdyanov  staff  hello.txt
lrwxrwxr-x  1 i.simdyanov  staff  newhello.txt -> hello.txt
drwxrwxr-x  2 i.simdyanov  staff  test
```

Здесь первый символ может принимать одно из значений:

- `-` — обычный файл;
- `l` — символическая ссылка;
- `d` — каталог.

Далее в девяти следующих символах записаны права владельца (первые три символа), группы (вторые три символа) и всех остальных (третьи три символа). Как можно видеть, под каждый из типов пользователей отводится по три символа:

- `r` — чтение;
- `w` — запись;
- `x` — запуск.

Так, последовательность `rwx` означает, что заданы все права. Если вместо каких-либо символов стоят прочерки, это означает, что такие права не заданы. Например, последовательность `rw-` означает права на чтение и запись, а `r--` — только на чтение.

В отчете команды `ls -la` можно увидеть владельца файла `i.simdyanov` и группу `staff`. Пользователи, которые не являются владельцами и не входят в группу, рассматриваются как все остальные пользователи.

Права доступа могут быть заданы в виде восьмеричного числа. Чтение задается цифрой 4, запись — 2, а исполнение — 1 (рис. 28.4).

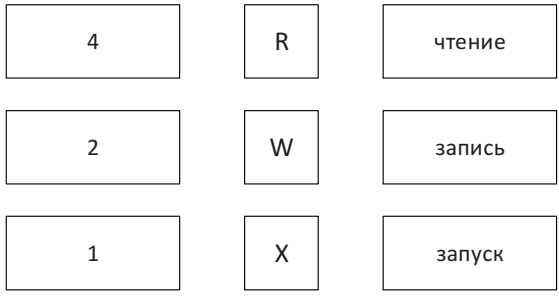


Рис. 28.4. Кодирование прав доступа

Для разрешения чтения (4) и записи в файл (2) используется цифра 6 (2 + 4 = 6), для предоставления полного доступа — цифра 7 (1 + 2 + 4 = 7). В результате права доступа образуют трехзначное восьмеричное число, первое значение в котором относится к владельцу файла, второе — к группе, третье — ко всем остальным.

Для файлов наиболее приемлемые права доступа: чтение и запись — для владельца и чтение — для всех остальных, т. е. 644. Для того чтобы иметь возможность «заходить» в каталог, для них необходимо задавать права доступа и на исполнение тоже, поэтому для каталогов следует выставлять права доступа 755.

> ***Замечание***
>
> В командной строке и в языке Ruby восьмеричные числа начинают с нуля: 0644 или 0755.

Чтобы изменить права доступа, можно воспользоваться командой `chmod`:

```
$ ls -la
-rw-rw-r--@  1 i.simdyanov  staff    13 28 окт  2019 hello.txt
$ chmod 0640 hello.txt
$ ls -la
-rw-r-----@  1 i.simdyanov  staff    13 28 окт  2019 hello.txt
```

Файлы, которые снабжаются правами доступа `x`, могут запускаться на выполнение. По умолчанию командная оболочка пытается запустить файл командой `bash`. Для того чтобы файл выполнялся при помощи интерпретатора Ruby, в первую строчку файла потребуется добавить специальный комментарий: ши-бенг (листинг 28.7).

Листинг 28.7. Использование ши-бенга. Файл execute.rb

```
#!/usr/bin/env ruby
puts 'Hello, world!'
```

Ши-бенг состоит из символа решетки и следующего за ним восклицательного знака. Затем указывается команда, при помощи которой будет запускаться файл. В приведенном примере мы воспользовались командой `env`, которая ищет интерпретатор Ruby, после чего командная оболочка будет знать, при помощи какой команды следует запускать содержимое файла.

Теперь можно назначить файлу права доступа на запуск — например, 0755:

```
$ chmod 0755 execute.rb
$ ls -la
-rwxr-xr-x@  1 i.simdyanov  staff    41 27 янв 15:47 execute.rb
$ ./execute.rb
Hello, world!
```

Класс `File` предоставляет одноименный метод `chmod` для изменения прав доступа файла (листинг 28.8).

Листинг 28.8. Изменение прав доступа из Ruby-программы. Файл chmod.rb

```
File.chmod(0755, 'chmod.rb')
```

Метод `chmod` можно применять в том числе и к открытому файлу.

При запуске Ruby-программы процессу так же назначаются пользователь и группа. Именно они используются для проверки, имеет ли программа права доступа к файлу. Если прав недостаточно, возникает исключительная ситуация (см. листинг 28.6).

Проверить, имеет ли программа права на чтение, запись и запуск файла, можно при помощи следующих методов класса `File`:

- `readable?` — возвращает `true`, если программа имеет право на чтение файла, иначе возвращается `false`;
- `writable?` — возвращает `true`, если программа имеет право на запись файла, иначе возвращается `false`;
- `executable?` — возвращает `true`, если программа имеет право на запуск файла, иначе возвращается `false`.

В листинге 28.9 демонстрируется использование методов для проверки права доступа файла.

Листинг 28.9. Проверка прав доступа. Файл chmod_check.rb

```
p File.readable?('chmod.rb')       # true
p File.writable?('chmod.rb')       # true
p File.executable?('chmod.rb')     # true

p File.readable?('/etc/sudoers')   # false
p File.writable?('/etc/sudoers')   # false
p File.executable?('/etc/sudoers') # false
```

Задания

1. Создайте скрипт, который выводит список каталогов и вложенных подкаталогов текущего каталога.
2. Создайте скрипт, который для всех файлов текущего каталога изменяет права доступа на 0644.

ГЛАВА 29

Каталоги

Файлы с исходными кодами этой главы находятся в каталоге *catalogs* сопровождающего книгу электронного архива.

Каталоги — специальный тип файлов, которые могут содержать ссылки на другие файлы и каталоги. Однако редактировать такие файлы непосредственно невозможно — это задача операционной системы.

Ruby предоставляет разработчикам специальный класс `Dir`, который позволяет создавать, удалять каталоги, а также читать их содержимое.

29.1. Текущий каталог

При запуске Ruby-программы ей назначается *текущий каталог*. Именно относительно этого каталога отсчитываются относительные пути, именно в нем создаются файлы при использовании метода `File.open`. Узнать путь до текущего каталога можно разными способами (листинг 29.1).

Листинг 29.1. Текущий каталог. Файл current_catalog.rb

```
puts Dir.pwd                                  # /home/igor/catalogs"
puts Dir.getwd                                # /home/igor/catalogs"
puts File.dirname(File.expand_path(__FILE__)) # /home/igor/catalogs"
```

Методы `pwd` и `getwd` являются синонимами. Можно использовать предопределенную константу `__FILE__`, возвращающую имя текущего Ruby-файла, получить абсолютный путь к нему при помощи метода `expand_path` и отсечь имя файла при помощи метода `dirname`.

Изменить текущий каталог можно при помощи метода `chdir` класса `Dir`. В листинге 29.2 текущий каталог изменяется на каталог, уровнем выше. Для этого методу `chdir` передается последовательность `..`, которая обозначает родительский каталог.

Листинг 29.2. Смена текущего каталога. Файл chdir.rb

```
puts Dir.pwd     # /home/igor/catalogs"
puts __dir__     # /home/igor/catalogs"

Dir.chdir('..')

puts Dir.pwd     # /home/igor"
puts __dir__     # /home/igor/catalogs"
```

Метод `__dir__` указывает на каталог, в котором расположен Ruby-файл. Значение, которое возвращает метод, не изменяется после смены текущего каталога.

29.2. Создание каталога

Для создания каталогов в командной оболочке предназначена команда `mkdir`, которая принимает в качестве аргумента имя нового каталога:

```
$ mkdir test
```

Класс `Dir` предоставляет одноименный метод. В листинге 29.3 создается каталог `test` (листинг 29.3).

Листинг 29.3. Создание нового каталога. Файл mkdir.rb

```
Dir.mkdir('test')
```

Метод `mkdir` может создать только один каталог — если ему указать цепочку несуществующих каталогов, возникнет исключительная ситуация (листинг 29.4).

Листинг 29.4. Метод `mkdir` не может создать несколько каталогов. Файл mkdir_fail.rb

```
Dir.mkdir('hello/world')
```

Запустив эту программу на выполнение, получаем сообщение об ошибке:

```
$ ruby mkdir_fail.rb
mkdir_fail.rb:1:in `mkdir': No such file or directory @ dir_s_mkdir - hello/
                                                          world (Errno::ENOENT)
```

В командной оболочке выходом из ситуации будет использование параметра `-p` команды `mkdir`:

```
$ mkdir -p hello/world
```

В Ruby для решения проблемы придется воспользоваться классом `FileUtils` и его методом `mkdir_p` (листинг 29.5).

Листинг 29.5. Использование метода `FileUtils.mkdir_p`. Файл mkdir_p.rb

```
require 'fileutils'
FileUtils.mkdir_p('hello/world')
```

29.3. Чтение каталога

Как уже отмечалось ранее, каталоги — это специальные файлы файловой системы, которые содержат список других файлов и подкаталогов. Поэтому порядок работы с ними такой же, как с файлами: открываем, читаем, закрываем.

Чтобы открыть каталог, необходимо создать объект класса `Dir`, для чего можно воспользоваться методом `new`. В листинге 29.6 открывается текущий каталог и с помощью метода `entries` извлекается его содержимое. Метод возвращает подкаталоги и файлы в виде массива.

Листинг 29.6. Открытие каталога. Файл dir_new.rb

```
d = Dir.new('.')
p d.entries # [".", "..", ..., "entries.rb"]
d.close
```

После завершения работы с каталогом его желательно закрыть при помощи метода `close`. Так же, как и в случае объектов класса `File`, можно воспользоваться методом `open`, который принимает блок. Внутри блока можно выполнять любые действия с каталогом, а после выхода из блока каталог будет автоматически закрыт (листинг 29.7).

Листинг 29.7. Открытие каталога при помощи метода `Dir.open`. Файл open.rb

```
Dir.open('.') do |d|
  p d.entries # [".", "..", ..., "entries.rb"]
end
```

В результирующем массиве, помимо файлов и подкаталогов, присутствуют ссылки на текущий `.` и родительский `..` каталоги.

Метод `entries` можно использовать без открытия каталога, правда, в этом случае методу следует передать в качестве параметра путь к каталогу (листинг 29.8).

Листинг 29.8. Использование метода `entries`. Файл entries.rb

```
p Dir.entries '..' # [".", "..", "arrays", ...]
```

Класс `Dir` включает модуль `Enumerable`, поэтому с его объектами можно обращаться как с коллекциями:

```
 > Dir.ancestors
=> [Dir, Enumerable, Object, Kernel, BasicObject]
```

Обойти содержимое каталога можно при помощи итератора `each` (листинг 29.9).

Листинг 29.9. Обход каталога при помощи метода `each`. Файл each.rb

```
Dir.open('.') do |d|
  d.each { |f| puts "#{f} => #{File.file?(f) ? 'файл' : 'каталог'}" }
end
```

Впрочем, для обхода содержимого каталога класс `Dir` предоставляет метод `foreach` (листинг 29.10).

Листинг 29.10. Обход каталога при помощи метода `foreach`. Файл foreach.rb

```
Dir.foreach('.') do |f|
  puts "#{f} => #{File.file?(f) ? 'файл' : 'каталог'}"
end
```

Используя итератор `reduce`, можно подсчитать размер файлов в текущем каталоге. В листинге 29.11 при помощи метода `entries` извлекается содержимое каталога, при помощи итератора `reject` отбрасываются все подкаталоги и, наконец, при помощи итератора `reduce` подсчитывается суммарный размер всех файлов в каталоге.

Листинг 29.11. Подсчет размера файлов в каталоге. Файл reduce.rb

```
p Dir.entries('.')
     .reject{ |f| File.directory? f }
     .reduce(0) { |total, f| total + File.size(f) }
```

29.4. Фильтрация содержимого каталога

Многие утилиты командной строки допускают использование *шаблонов*. При выводе содержимого каталога с помощью команды `ls` для задания произвольного количества символов можно использовать символ `*`. Так, шаблон `*.rb` позволяет отобрать только файлы с расширением rb:

```
$ ls -la *.rb
-rw-r--r--@ 1 i.simdyanov  staff  285 27 янв 17:12 chdir.rb
-rw-r--r--@ 1 i.simdyanov  staff  207 27 янв 17:11 current_catalog.rb
-rw-r--r--@ 1 i.simdyanov  staff   70 27 янв 18:26 dir_new.rb
...
```

Знак вопроса `?` означает один любой символ:

```
$ ls -la mkdir_?.rb
-rw-r--r--@ 1 i.simdyanov  staff  54 27 янв 17:48 mkdir_p.rb
```

Утилита `ls` не единственная, которая поддерживает шаблоны. Их поддерживает также утилита для удаления файлов `rm`:

```
$ rm *.txt
```

Точно такие же шаблоны можно использовать в Ruby и для фильтрации. Для этого шаблоны передаются квадратным скобкам класса `Dir` (листинг 29.12).

Листинг 29.12. Фильтрация содержимого каталога. Файл dir_bracket.rb

```
p Dir['*.rb'] # ["chdir.rb", ...]
```

Вместо квадратных скобок можно использовать метод класса `glob` (листинг 29.13).

Листинг 29.13. Использование метода `glob`. Файл glob.rb

```
p Dir.glob('*') # ["chdir.rb", ...]
```

29.5. Рекурсивный обход каталога

Простейшие средства обхода каталога выводят только содержимое каталога, не заглядывая в подкаталоги (рис. 29.1).

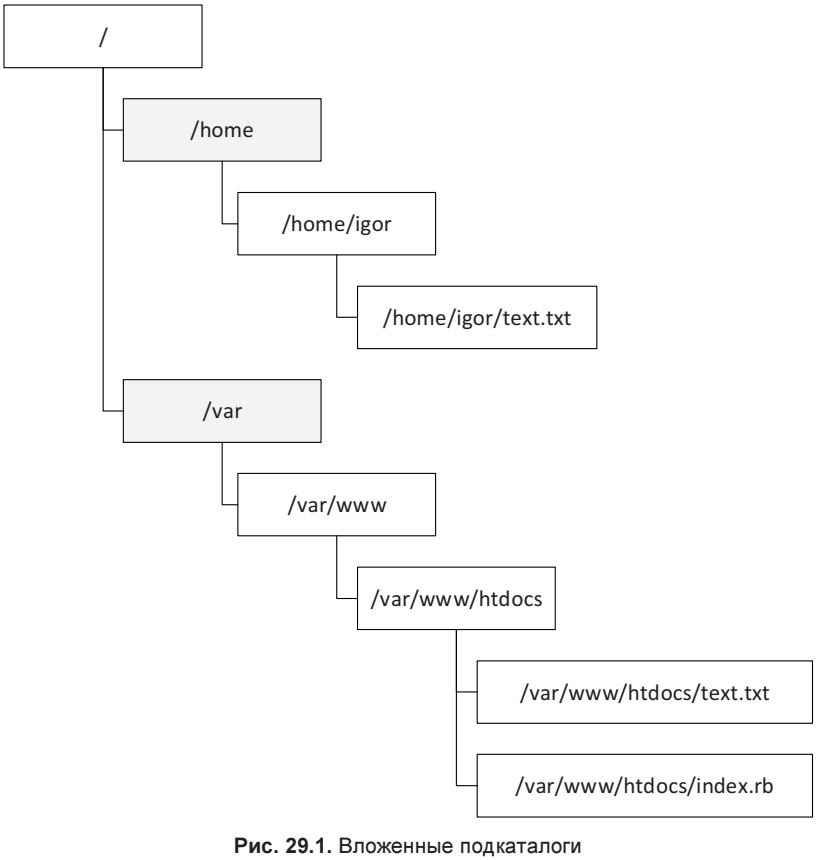


Рис. 29.1. Вложенные подкаталоги

Для того чтобы обойти все вложенные подкаталоги, потребуется *рекурсивный спуск*. То есть такой метод, который при встрече на очередной итерации каталога будет вызывать сам себя.

Давайте разработаем программу, которая подсчитывает количество Ruby-файлов в каталоге с учетом всех вложенных подкаталогов. Разработку такой программы можно начать со сканирования текущего каталога, который обозначается точкой (листинг 29.14).

Листинг 29.14. Сканирование текущего каталога. Файл scan.rb

```
path = File.join('.')
p Dir.new(path).entries.reject { |x| %w[. ..].include? x }
```

Здесь мы с помощью итератора `reject` избавляемся от каталогов `.` и `..`, которые указывают на текущий и родительский каталоги.

Извлечь расширение из пути к файлу можно при помощи метода `extname` класса `File`. В листинге 29.15 переменная `counter` увеличивается на единицу всякий раз, когда элемент текущего каталога имеет расширение rb.

Листинг 29.15. Подсчет файлов с расширением rb. Файл extname.rb

```
path = File.join('.')
entries = Dir.new(path).entries.reject { |x| %w[. ..].include? x }

counter = 0
entries.each do |item|
  counter += 1 if File.extname(item) == '.rb'
end
p counter
```

Полученный скрипт не умеет заглядывать в подкаталоги, поэтому необходимо научить его повторно сканировать подкаталог, если метод `File.directory?` возвращает `true` для текущего элемента:

```
...
entries.each do |item|
  counter += ... if File.directory?(item)
  counter += 1 if File.extname(item) == '.rb'
end
...
```

Для этого необходимо оформить полученный код в рекурсивный метод (листинг 29.16).

Листинг 29.16. Рекурсивный метод. Файл recuresive.rb

```
def scan(path)
  entries = Dir.new(path)
              .entries
```

```
              .reject { |x| %w[. ..].include? x }
              .map { |x| File.join(path, x) }

  counter = 0
  entries.each do |item|
    counter += scan(item) if File.directory?(item)
    counter += 1 if File.extname(item) == '.rb'
  end
  counter
end

path = File.join('.')
p scan(path)
```

Элементы массива `entries` необходимо превратить в относительные пути, т. к. текущий каталог остается неизменным. Для этого элементы массива обходятся при помощи метода `map`, в котором методом `File.join` к текущему пути `path` прибавляется текущий элемент массива `entries`. Это позволяет формировать пути вида:

```
./hello/world/index.rb
./hello/world
./hello
./mkdir.rb
```

В случае, если по пути метода встречается каталог, для него повторно вызывается метод `scan`. В качестве результата метод `scan` возвращает количество подсчитанных элементов. Это значение будет прибавляться к счетчику `counter` на каждой итерации рекурсии.

Запуск программы будет подсчитывать в каталоге все Ruby-файлы с учетом вложенных подкаталогов.

Вариант программы из листинга 29.16 можно усовершенствовать. Например, можно отказаться от итератора `each` и использовать `reduce` (листинг 29.17). Это позволит избавиться от явной инициализации переменной `counter`.

Листинг 29.17. Альтернативная реализация программы. Файл scan_reduce.rb

```
def scan(path)
  entries = Dir.new(path)
              .entries
              .reject { |x| %w[. ..].include? x }
              .map { |x| File.join(path, x) }

  entries.reduce(0) do |counter, item|
    counter += scan(item) if File.directory?(item)
    counter += (File.extname(item) == '.rb') ? 1 : 0
  end
end
```

```
path = File.join('.')
p scan(path)
```

29.6. Удаление каталога

Для удаления каталога можно воспользоваться методом `rmdir` (листинг 29.18). Удалять можно только те каталоги, в которых отсутствуют файлы и вложенные подкаталоги.

Листинг 29.18. Удаление каталога. Файл rmdir.rb

```
Dir.rmdir('test')
```

Для того чтобы удалить не пустой каталог, нужно рекурсивно спуститься по всем его подкаталогам и удалить все вложенные подкаталоги и файлы.

Задания

1. Создайте метод рекурсивного удаления содержимого каталога со всеми вложенными подкаталогами.
2. Пусть имеется каталог с файлами-изображениями. Создайте программу, которая выбирает случайным образом одно изображение. Другие файлы при этом должны игнорироваться.
3. Создайте метод, который выводит список подкаталогов в текущем каталоге. Рядом с каждым из подкаталогов следует вывести список файлов в нем (с учетом вложенных подкаталогов).
4. Создайте программу подсчета количества строк в файлах проекта. Она должна обходить все вложенные каталоги проекта и подсчитывать количество строк в файлах с определенными расширениями — например, *.rb.
5. Создайте программу, которая бы осуществляла обход вложенных подкаталогов текущего каталога и заменяла бы во всех файлах одну подстроку другой.
6. Создайте скрипт, который копировал бы содержимое одного каталога вместе со всеми вложенными подкаталогами в другой.
7. Создайте скрипт, который подсчитывает объем, занимаемый файлами каталога, включая файлы всех вложенных подкаталогов. Выведите размер в байтах, килобайтах, мегабайтах или гигабайтах в зависимости от полученного объема.

ГЛАВА 30

Регулярные выражения

Файлы с исходными кодами этой главы находятся в каталоге *regexp* сопровождающего книгу электронного архива.

Регулярные выражения — это специализированный декларативный язык для описания и поиска фрагментов в тексте. Они позволяют искать, заменять, извлекать тексты по правилам произвольной сложности.

По сравнению со строковыми методами, регулярные выражения значительно компактнее. Там, где требуются десятки строк Ruby-кода, порой достаточно одной строки регулярного выражения.

Как и любой декларативный язык, регулярные выражения весьма сложны. Чтобы их освоить, придется специально изучать их синтаксис и тренироваться.

Регулярные выражения реализованы практически в каждом современном языке программирования. Существуют два главных диалекта: Perl- и POSIX-регулярные выражения. Несмотря на то, что POSIX — это официальный стандарт, за последние 30 лет де-факто стандартом стал Perl-вариант. Ларри Уолл, создатель языка Perl, подобрал настолько удачную форму регулярных выражений, что они были заимствованы практически всеми современными языками программирования.

Впрочем, каждый язык немного изменяет и адаптирует регулярные выражения. Ruby-вариант регулярных выражений также не является исключением — он немного отличается от их реализации в других языках программирования.

30.1. Как изучать регулярные выражения?

Объемные и сложные задачи можно решать долго при помощи простых методов или быстро — при помощи сложных. Классические строковые методы легко осваиваются, но требуют длительного времени при решении более или менее сложных повседневных задач. Регулярные выражения освоить гораздо сложнее, и времени на это уходит больше, однако повседневные задачи, даже достаточно сложные, они позволяют решать в короткий срок и с минимальными усилиями.

Куда потратить время — каждый из разработчиков решает сам. Однако свободное владение регулярными выражениями дает в программировании ту же свободу дей-

ствий, что и десятипальцевый слепой метод в наборе текста. Вы будете излагать свои мысли в коротких и элегантных строках регулярных выражений, а не тратить десятки часов на составление и отладку объемных блоков кода. Полученные знания можно применить в десятке программных систем. Это и языки программирования (Python, Perl, C#, Java, PHP, JavaScript и т. п.), и базы данных (PostgreSQL, MySQL, Oracle), и командная строка.

Регулярные выражения — декларативный язык программирования, требующий от разработчиков высокой сосредоточенности, особенно на начальных этапах обучения. В отличие от объектно-ориентированного языка, в декларативном программировании мы не составляем алгоритм, последовательность шагов и состояний, через которые нужно пройти программе. Вместо этого задается конечная цель — результат, которого мы хотим достигнуть. Компьютер самостоятельно ищет путь для решения задачи. Если все работает как задумано, это выглядит волшебством — по короткому требованию компьютер решает трудную задачу. Однако, если что-то идет не так, отладка становится необычайно сложной, т. к. все шаги компьютера скрыты под «капотом» декларативного языка. Необходимо хорошо представлять, как механизм регулярных выражений устроен внутри, чтобы вскрыть ошибку или модифицировать поведение уже составленного регулярного выражения.

Так как регулярные выражения весьма сложны, ими стараются не злоупотреблять. Часть команды разработчиков может не владеть ими, поэтому сопровождать и модифицировать код становится труднее. Даже если регулярные выражения хорошо знакомы всем участникам команды, их разбор и модификация требуют сравнительно много времени. Чем больше регулярное выражение — тем выше вероятность ошибиться. Поэтому вместо одного большого регулярного выражения лучше использовать несколько коротких.

Несмотря на то, что в этой главе детально обсуждаются синтаксис и использование регулярных выражений, полностью рассмотреть их здесь невозможно. Для более глубокого изучения материала следует обратиться к книге Дж. Фридла «Регулярные выражения».

30.2. Синтаксический конструктор

С точки зрения Ruby, регулярные выражения — это объекты класса `Regexp`. Создать объект этого класса можно как с использованием метода `new`, так и при помощи синтаксического конструктора в виде двух слешей: `//`. Кроме того, регулярные выражения можно создать при помощи конструктора `%r` (листинг 30.1).

Листинг 30.1. Создание регулярного выражения. Файл new.rb

```
p Regexp.new('') # //
p //             # //
p %r()           # //
```

На практике вызов `Regexp.new` практически никогда не встречается. Дело в том, что метод `new` принимает в качестве аргумента строку. Как в строке, так и в регулярных

выражениях, интенсивно используется экранирование при помощи символа обратного слеша \ — в регулярных выражениях задаются специальные символы, а у специальных символов, наоборот, отменяется специальное поведение.

В случае `Regexp.new` символ обратного слеша в строках приходится дополнительно экранировать. В случае синтаксического конструктора `//` дополнительного экранирования можно избежать (листинг 30.2).

Листинг 30.2. Экранирование в строках. Файл escape.rb

```
p Regexp.new('\\\\') # /\\/
p /\\/               # /\\/
p %r(\\)             # /\\/
```

Как видно из приведенного примера, содержимое регулярного выражения размещается между двумя слешами.

30.3. Оператор =~

Воспользоваться регулярным выражением можно множеством способов. Один из самых простых — это использовать оператор `=~`, который проверяет, находит ли регулярное выражение хотя бы одно соответствие в строке.

В листинге 30.3 представлено простейшее регулярное выражение `/Ruby/`. Строки можно рассматривать как массивы символов, проиндексированных с нуля (см. *разд. 4.2.4*). Оператор `=~` возвращает позицию первого найденного соответствия. Если регулярное выражение не обнаружило совпадений, возвращается неопределенное значение `nil`.

Листинг 30.3. Использование оператора =~. Файл equal.rb

```
p 'Hello, Ruby! Ruby is cool!' =~ /Ruby/ # 7
p 'Язык программирования Ruby' =~ /Ruby/ # 22
p /Ruby/ =~ 'Язык программирования Ruby' # 22
p 'Hello world' =~ /Ruby/ # nil
```

Регулярное выражение может находиться как слева, так и справа от оператора `=~` (листинг 30.4).

Листинг 30.4. Симметричность оператора =~. Файл equal_sym.rb

```
p 'Hello, Ruby!' =~ /Ruby/ # 7
p /Ruby/ =~ 'Hello, Ruby!' # 7
```

Строки поддерживают квадратные скобки, при помощи которых можно извлечь произвольный фрагмент (см. *разд. 4.2.5*). В листинге 30.5 приводится несколько примеров извлечения подстрок при помощи квадратных скобок.

Листинг 30.5. Использование квадратных скобок. Файл string_bracket.rb

```
str = 'Hello, world!'
puts str[7, 5] # world
puts str[7..11] # world
```

Помимо числовых значений и диапазонов, квадратные скобки допускают использование регулярных выражений (листинг 30.6).

Листинг 30.6. Регулярные выражения в квадратных скобках. Файл string_regexp.rb

```
str = 'Hello, world!'
p str[/world/] # world
p str[/\w+!/]  # world!
```

30.4. Методы поиска

Для работы регулярными выражениями предусмотрено несколько методов: `match`, `match?`, `scan`. Ряд из них реализован как в классе `Regexp`, так и в классе `String`. Часть их — например, `scan`, — реализована только в классе `String`.

30.4.1. Метод *match*

Вместо оператора `=~` для поиска соответствия регулярному выражению можно использовать метод `match` (листинг 30.7). Результатом работы метода является объект класса `MatchData`.

Листинг 30.7. Поиск соответствий при помощи метода `match`. Файл match.rb

```
str = 'Hello, Ruby! Ruby is cool!'
m = str.match /Ruby/

p m      # #<MatchData "Ruby">
p m.to_a # ["Ruby"]
```

Метод `match` возвращает неопределенное значение `nil`, если соответствие регулярному выражению не обнаружено.

Метод `match` реализован в классах `Regexp` и `String`, поэтому регулярное выражение может быть как получателем, так и аргументом (листинг 30.8).

Листинг 30.8. Регулярное выражение может быть и слева, и справа. Файл match_reg.rb

```
str = 'Hello, Ruby! Ruby is cool!'
reg = /Ruby/

p str.match reg
p reg.match str
```

Для доступа к результатам можно использовать квадратные скобки (листинг 30.9).

Листинг 30.9. Доступ к результатам. Файл match_bracket.rb

```
m = 'Hello, Ruby! Ruby is cool!'.match /Ruby/
puts m[0] # Ruby
```

Чтобы получить результат, внутри квадратных скобок здесь указывается индекс `0`. Допускается также использование других индексов и даже символьных ключей, как в хэшах. Однако для этого необходимо использовать группировку при помощи круглых скобок или именованные группы, с которыми мы познакомимся в *разд. 30.5.1*.

По умолчанию регулярные выражения ищут соответствия слева направо, и как только соответствие обнаружено, метод `match` прекращает свою работу (рис. 30.1).

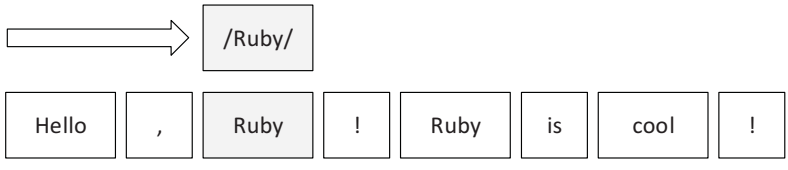


Рис. 30.1. Поиск соответствия методом `match`

Метод `match` может принимать второй необязательный аргумент. Через него можно задать в строке позицию, начиная с которой в ней будет проводиться поиск. Забегая вперед, задействуем регулярное выражение `\d+` для поиска в строке числа (листинг 30.10). Регулируя позицию начала поиска, мы сможем обнаружить в строке все числа.

Замечание

Более детально синтаксис регулярных выражений будет рассмотрен в *разд. 30.5*.

Листинг 30.10. Использование второго аргумента `match`. Файл match_second_arg.rb

```
str = 'Цена билета увеличилась с 500 до 600 рублей'
p str.match(/\d+/)     #<MatchData "500">
p str.match(/\d+/, 30) #<MatchData "600">
```

30.4.2. Метод *match?*

Логический метод `match?` действует точно так же, как и метод `match`. Более того, он может принимать в том числе второй необязательный параметр для поиска с заданной позиции, а не с начала строки.

Однако, вместо объекта класса `MatchData`, метод возвращает либо `true` — если соответствие найдено, либо `false` — если совпадение с регулярным выражением не обнаружено (листинг 30.11).

Замечание

Метод `match?` добавлен в Ruby, начиная с версии 2.4.0.

Листинг 30.11. Использование метода `match?`. Файл match_logic.rb

```
str = 'Hello, Ruby!'

p str.match? /Ruby/  # true
p str.match? /world/ # false
```

30.4.3. Метод *scan*

Метод `match` находит лишь первое соответствие регулярному выражению, и если необходимо обнаружить все подстроки, следует воспользоваться методом `scan` (листинг 30.12). Метод `scan` возвращает массив со всеми обнаруженными вхождениями регулярного выражения в строку.

Листинг 30.12. Использование метода `scan`. Файл scan.rb

```
m = 'Hello, Ruby! Ruby is cool!'.scan /Ruby/
p m      # ["Ruby", "Ruby"]
p m.size # 2
```

30.5. Синтаксис регулярных выражений

Продемонстрированные до этого момента примеры не очень впечатляющие. Связано это с тем, что мы практически не используем возможности регулярных выражений.

30.5.1. Метасимволы

В выражении `/Ruby/` все символы обозначают сами себя — мы ищем вхождение подстроки `'Ruby'` в строку. Однако ряд символов внутри регулярного выражения имеет особый смысл. В табл. 30.1 представлены метасимволы регулярных выражений.

Таблица 30.1. *Метасимволы регулярных выражений*

Метасимвол	Описание
`.`	Один любой символ
`[...]`	Один символ из тех, что указаны в квадратных скобках. Например, выражение `[0123456789]` соответствует одной цифре, и оно может быть свернуто в диапазон `[0-9]`

Таблица 30.1 *(окончание)*

Метасимвол	Описание
`[^...]`	Один любой символ, который не соответствует диапазону в квадратных скобках. Например, выражение `[^0-9]` соответствует любому символу, не являющемуся цифрой
`^ $ \A \z`	Границы строк, начало или окончание
`|`	Логическое ИЛИ, используется совместно с круглыми скобками `(world|Ruby)`
`(...)`	Группировка символов

Символ «точка» обозначает один любой символ. В листинге 30.13 в строках `'hello'` и `'world'` при помощи регулярного выражения ищется первый символ `l` и следующий за ним любой символ.

Листинг 30.13. Поиск любого символа. Файл search_any.rb

```
regexp = /l./

p 'Hello'.match regexp #<MatchData "ll">
p 'world'.match regexp #<MatchData "ld">
```

При помощи квадратных скобок можно указывать диапазоны символов (листинг 30.14).

Листинг 30.14. Поиск любого символа из диапазона. Файл search_bracket.rb

```
p '500'.match /[56]/ #<MatchData "5">
p '600'.match /[56]/ #<MatchData "6">
p '700'.match /[56]/ # nil
```

Регулярное выражение находит соответствие в числе `'500'` и `'600'`, однако в числе `'700'` подходящей подстроки уже не обнаруживается.

Внутри квадратных скобок можно использовать диапазоны символов. Например, последовательность `[0-9]` соответствует любой цифре от 0 до 9 (листинг 30.15).

Листинг 30.15. Поиск любой цифры. Файл search_numbers.rb

```
p '700'.match /[0-9]/ #<MatchData "7">
```

При помощи символа ^ внутри квадратных скобок можно задавать отрицание. Например, выражение `[^0-9]` означает любой символ за исключением цифр (листинг 30.16).

Листинг 30.16. Поиск любого символа, кроме цифр. Файл search_not_numbers.rb

```
p '700p'.match /[^0-9]/ #<MatchData "p">
```

Регулярное выражение может содержать не только символы, но и позиции в строке. Например, при помощи символа ^ можно привязаться к началу строки (листинг 30.17).

Листинг 30.17. Привязка к началу строки при помощи ^. Файл string_begin.rb

```
p 'Hello, world!'.match /world/  #<MatchData "world">
p 'Hello, world!'.match /^world/ # nil
p 'world of Ruby'.match /^world/ #<MatchData "world">
```

Здесь регулярное выражение /^world/ находит соответствие, только если слово 'world' расположено в начале строки.

Можно привязываться к концу строки при помощи символа $ (листинг 30.18).

Листинг 30.18. Привязка к концу строки при помощи $. Файл string_end.rb

```
p 'Hello, Ruby!'.match /Ruby/   #<MatchData "Ruby">
p 'Hello, Ruby!'.match /Ruby$/  # nil
p 'world of Ruby'.match /Ruby$/ #<MatchData "Ruby">
```

Следует внимательно относиться к многострочным выражениям. Пусть имеется строка, в которой слова 'hello' и 'world' расположены на отдельных строках. Привязка к концу строки при помощи символов ^ и $ реагирует не только на окончание строки, но и на перевод внутри строки (листинг 30.19).

Листинг 30.19. Использование последовательностей \A и \z. Файл string_multiline.rb

```
str = <<~here
        hello
        world
      here

p str                 # "hello\nworld\n"
p str.match /hello$/  #<MatchData "hello">
p str.match /hello\z/ # nil

p str.match /^world/  #<MatchData "world">
p str.match /\Aworld/ # nil
p str.match /\Ahello/ #<MatchData "hello">
```

Если необходимо привязаться к началу или окончанию всей строки без учета внутренних переводов строки, следует использовать последовательности \A и \z (см. листинг 30.19).

При помощи символа | можно задать в регулярных выражениях логику ИЛИ. Пусть имеется фразы:

```
'Hello, world!'
'Hello, ruby!'
'Hello, Igor!'
```

Напишем регулярное выражение, которое будет соответствовать строкам, в которых встречается или слово `'world'`, или слово `'ruby'`. Для этого можно воспользоваться символом `|` (листинг 30.20).

Листинг 30.20. Использование метасимвола `|`. Файл pipe.rb

```
regexp = 'Hello, (world|ruby)!'

p 'Hello, world!'.match regexp #<MatchData "Hello, world!" 1:"world">
p 'Hello, ruby!'.match regexp  #<MatchData "Hello, ruby!" 1:"ruby">
p 'Hello, Igor!'.match regexp  # nil
```

Для группировки вариантов, разделенных символом `|`, мы здесь использовали круглые скобки. Их можно использовать и без символа `|`. В этом случае они помечают фрагменты регулярного выражения, которые потом можно извлечь из результирующего объекта.

Объект класса `MatchData` ведет себя как массив и допускает вызов квадратных скобок, в которых указываются индексы. Индекс `0` соответствует полному регулярному выражению, `1` — фрагменту в первых круглых скобках, `2` — второму и т. д. (листинг 30.21).

Листинг 30.21. Использование круглых скобок. Файл bracket.rb

```
regexp = 'Hello, (world)!'
result = 'Hello, world!'.match regexp

p result.to_a  # ["Hello, world!", "world"]
puts result[0] # Hello, world!
puts result[1] # world
```

Если необходимо извлечь содержимое только круглых скобок, можно воспользоваться методом `scan` (листинг 30.22).

Листинг 30.22. Извлечение соответствий при помощи `scan`. Файл bracket_scan.rb

```
regexp = /(Ruby)/
p 'World of Ruby! Hello, Ruby!'.scan(regexp) # [["Ruby"], ["Ruby"]]
```

Метод `scan` возвращает массив массивов, т. к. он может обнаруживать несколько вариантов, подходящих под регулярное выражение (листинг 30.23).

Листинг 30.23. Файл bracket_scan_many.rb

```
regexp = /(Hello), (world|Ruby)!/
result = 'Hello, world! Hello, Ruby!'.scan(regexp)
p result # [["Hello", "world"], ["Hello", "Ruby"]]
```

Значения в круглых скобках можно именовать. Для этого сразу после открывающей круглой скобки размещается знак вопроса, после которого в угловых скобках можно указать имя (листинг 30.24).

Листинг 30.24. Именованные круглые скобки. Файл bracket_named.rb

```
regexp = /Hello, (?<name>world)!/
result = 'Hello, world!'.match regexp

puts result[:name] # world
puts result['name'] # world
```

В случае именованных круглых скобок для доступа к результатам при помощи квадратных скобок вместо числовых индексов можно использовать символьные или строковые ключи.

Вместо угловых скобок для именованных фрагментов могут использоваться одиночные кавычки (листинг 30.25).

Листинг 30.25. Использование кавычек. Файл bracket_named_quotes.rb

```
regexp = /Hello, (?'name'world)!/
result = 'Hello, world!'.match regexp

puts result[:name]  # world
puts result['name'] # world
```

Иногда необходимо, чтобы содержимому круглых скобок не назначался индекс или имя. В этом случае сразу после открывающей круглой скобки следует поместить символы знака вопроса и двоеточия (листинг 30.26).

Листинг 30.26. Игнорирование круглых скобок. Файл bracket_ignore.rb

```
regexp = /(?:Hello), (world|Ruby)!/
result = 'Hello, world! Hello, Ruby!'.scan(regexp)
p result # [["world"], ["Ruby"]]
```

В круглых скобках вида `(?#...)` можно размещать комментарии (листинг 30.27).

Листинг 30.27. Комментарии. Файл bracket_comments.rb

```
regexp = /(?# Это комментарий)(?:Hello), (world|Ruby)!/
result = 'Hello, world! Hello, Ruby!'.scan(regexp)
p result # [["world"], ["Ruby"]]
```

30.5.2. Экранирование

Отменить специальное значение символа в регулярном выражении можно при помощи экранирования. Как и в строках, в регулярных выражениях символы экранируют обратным слешем \.

Например, символ «точка» является метасимволом и обозначает один любой символ. Если мы захотим найти при помощи регулярного выражения именно точку, нам придется отметить специальное значение точки при помощи экранирования (листинг 30.28).

Листинг 30.28. Экранирование символа точки. Файл escape_point.rb

```
regexp = /(Hello|world|Ruby)\./
p 'Hello, Ruby.'.match(regexp)  #<MatchData "Ruby." 1:"Ruby">
p 'Hello, world.'.match(regexp) #<MatchData "world." 1:"world">

regexp = /(Hello|world|Ruby)./
p 'Hello, Ruby.'.match(regexp)  #<MatchData "Hello," 1:"Hello">
p 'Hello, world.'.match(regexp) #<MatchData "Hello," 1:"Hello">
```

Чтобы найти точку, мы используем последовательность \., — если здесь точку не экранировать, она будет обозначать любой символ. В результате регулярное выражение будет находить подстроку 'Hello' и следующую за ней запятую.

При помощи экранирования ряду обычных символов можно назначать специальное значение. Например, если экранировать символ d, то последовательность \d будет обозначать любую цифры от 0 до 9, т. е. диапазон [0-9] (листинг 30.29).

Листинг 30.29. Экранирование символов $ и d. Файл escape_d.rb

```
p 'Цена $50'.match /\$\d\d/ #<MatchData "$50">
```

В приведенном примере все символы экранированы: у $ отменяется специальное назначение, для d оно, наоборот, добавляется. В табл. 30.2 представлены экранированные последовательности со специальным значением.

Таблица 30.2. *Специальные значения экранированных символов*

Метасимвол	Описание
\b	Граница слова
\t	Символ табуляции
\r	Символ возврата каретки
\n	Символ перевода строки
\d	Цифра [0-9]
\D	Символ, не являющийся цифрой

Таблица 30.2 *(окончание)*

Метасимвол	Описание
\w	Символ слова
\W	Символ, не являющийся частью слова
\s	Пробельный символ
\S	Символ, не являющийся пробельным
\h	Шестнадцатеричная цифра [0-9a-f]
\H	Символ, не являющийся шестнадцатеричной цифрой
\b	Граница слова
\B	Символ, не являющийся границей слова
\A	Начало строки
\z	Конец строки

30.5.3. Квантификаторы

Как видно из листинга 30.29, чтобы извлечь число 50, потребовалось два раза указать последовательность \d — под каждую из цифр. Если цена увеличится до 500 долларов, регулярное выражение по-прежнему будет извлекать только две цифры (листинг 30.30).

Листинг 30.30. Извлечение цены из строки. Файл price_match_wrong.rb

```
p 'Цена $500'.match /\$\d\d/ #<MatchData "$50">
```

Для того чтобы извлечь всю стоимость целиком, потребуется добавить третью последовательность \d. Однако регулярное выражение /\$\d\d\d/ уже не позволит извлекать цену $50. Для решения этой проблемы нужен механизм, позволяющий извлекать числа произвольной длины. Для этого предназначены *квантификаторы*, которые изменяют количество вхождений символов (табл. 30.3).

Таблица 30.3. *Квантификаторы*

Квантификатор	Описание
?	Одно или ноль вхождений
*	Ноль или несколько вхождений
+	Одно или несколько вхождений
{n}	Ровно n вхождений
{n,}	Больше n вхождений
{n,m}	От n до m вхождений

Квантификаторы действуют либо на предыдущий символ, либо на сгруппированное выражение, т. е. на круглые скобки. Регулярное выражение для поиска цены можно переписать с использованием квантификатора + или явно указав количество цифр в фигурных скобках (листинг 30.31).

Листинг 30.31. Использование квантификаторов. Файл price_match.rb

```
p 'Цена $500'.match /\$\d+/   #<MatchData "$500">
p 'Цена $500'.match /\$\d{3}/ #<MatchData "$500">
```

Выражения из приведенного примера уже не сработают для цены $50. Для решения этой задачи можно воспользоваться диапазоном (листинг 30.32).

Листинг 30.32. Использование диапазона. Файл price_match_range.rb

```
p 'Цена $500'.match /\$\d{1,3}/ #<MatchData "$500">
p 'Цена $50'.match /\$\d{1,3}/  #<MatchData "$50">
p 'Цена $5'.match /\$\d{1,3}/   #<MatchData "$5">

p 'Цена $500'.match /\$\d{3}/   #<MatchData "$500">
p 'Цена $50'.match /\$\d{3}/    # nil
p 'Цена $5'.match /\$\d{3}/     # nil
```

Последовательность \d можно помещать и внутрь квадратных скобок, более того, туда можно поместить и символ $ (листинг 30.33). Квадратные скобки рассматриваются как один символ, поэтому квантификаторы будут действовать и на них.

Листинг 30.33. Файл price_match_bracket.rb

```
p 'Цена $500'.match /[\$\d]+/ #<MatchData "$500">
```

Отрицание внутри квадратных скобок действует в том числе на последовательности вроде \d. В листинге 30.34 ищется часть строки, в которых нет чисел.

Листинг 30.34. Файл search_none_numbers.rb

```
p 'Цена билета 500 рублей'.match /[^\d]+/ #<MatchData "Цена билета ">
```

Пусть из строки необходимо извлечь первое слово. Для этого можно модифицировать регулярное выражение из предыдущего примера, добавив в диапазон пробельные символы (листинг 30.35).

Листинг 30.35. Извлечение слова. Файл search_world.rb

```
p 'Цена билета 500 рублей'.match /[^\s\d]+/ #<MatchData "Цена">
```

Здесь мы извлекаем любое количество символов, за исключением пробельных символов и цифр.

В качестве альтернативы для решения этой задачи можно указать диапазон букв русского алфавита (листинг 30.36).

Листинг 30.36. Диапазон букв русского алфавита. Файл search_world_alter.rb

```
p 'Цена билета 500 рублей'.match /[а-яёА-ЯЁ]+/ #<MatchData "Цена">
```

В этом примере мы выбираем любое количество символов, входящих в диапазон строчных (маленьких) и прописных (заглавных) букв русского алфавита. В силу особенностей кодирования буква ё в диапазон от «а» до «я» не входит, поэтому ее нужно указывать отдельно.

30.5.4. Опережающие и ретроспективные проверки

Помимо обычных круглых скобок, регулярные выражения предоставляют разработчику четыре типа позиционных проверок (табл. 30.4), позволяющих выяснить, находится та или иная подстрока справа или слева от текущей позиции. Если проверяется выражение слева, проверка называется *опережающей*, если справа — *ретроспективной*. Проверка на равенство называется *позитивной*, а проверка на неравенство — *негативной*.

Таблица 30.4. *Позиционные проверки*

Позиционная проверка	Описание
`(?<=...)`	Выражение в скобках ... располагается слева
`(?<!...)`	Выражение в скобках ... не может располагаться слева
`(?=...)`	Выражение в скобках ... располагается справа
`(?!...)`	Выражение в скобках ... не может располагаться справа

Позиционные проверки лишь проверяют, что слева или справа от выражения находится или отсутствует какая-то последовательность (рис. 30.2).

В листинге 30.37 приводятся примеры использования проверок. Несмотря на то, что содержимое проверок располагается в круглых скобках, захвата содержимого не происходит, и оно не попадает в результирующий `MatchData`-объект.

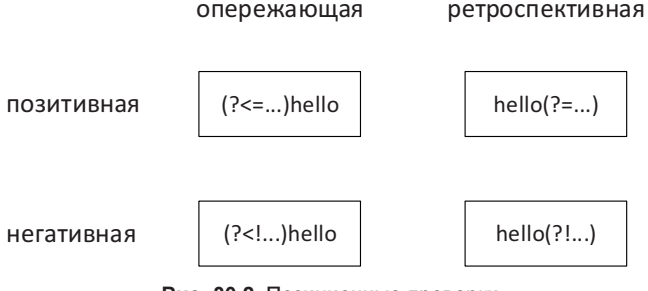


Рис. 30.2. Позиционные проверки

Листинг 30.37. Опережающие и ретроспективные проверки. Файл check.rb

```
p 'hello999'.match /(?<=hello)(9{3})/ #<MatchData "999" 1:"999">
p 'hello999'.match /(9{3})(?=hello)/  # nil

p 'hello999'.match /(?<!hello)(9{3})/ # nil
p 'hello999'.match /(9{3})(?!hello)/  #<MatchData "999" 1:"999">

p '999hello'.match /(?<=hello)(9{3})/ # nil
p '999hello'.match /(9{3})(?=hello)/  # <MatchData "999" 1:"999">

p '999hello'.match /(?<!hello)(9{3})/ # <MatchData "999" 1:"999">
p '999hello'.match /(9{3})(?!hello)/  # nil
```

30.6. Модификаторы

После синтаксического конструктора регулярного выражения // могут располагаться модификаторы. В листинге 30.38 приводится пример использования модификатора i, предписывающего игнорировать регистр.

Листинг 30.38. Использование модификаторов. Файл modifiers.rb

```
p Regexp.new('Ruby', Regexp::IGNORECASE) # /Ruby/i
p /Ruby/i                                # /Ruby/i
```

В табл. 30.5 приводится список модификаторов, которые допускается использовать после синтаксического конструктора.

Таблица 30.5. *Модификаторы регулярных выражений*

Модификатор	Описание
i	Игнорирование регистра, заглавные и маленькие буквы считаются эквивалентными
m	Мультистрочный режим, при котором соответствие ищется на нескольких строках. Если модификатор не указан, регулярное выражение ищет соответствие между двумя переводами строки
x	Игнорирование пробельных символов и комментариев внутри регулярного выражения
o	Режим вычисления интерполяции внутри регулярного выражения, при использовании модификатора интерполируемое выражение вычисляется только один раз

Регулярные выражения различают регистр: прописные (заглавные) и строчные (маленькие) буквы считаются разными. Отключить такое поведение можно при помощи модификатора i (листинг 30.39).

Листинг 30.39. Использование модификатора `i`. Файл modifier_i.rb

```
p /Ruby/.match 'Hello, ruby!'  # nil
p /Ruby/i.match 'Hello, ruby!' #<MatchData "ruby">
```

Мультистрока может содержать множество строк, отделенных друг от друга переводом строки `\n`. Регулярное выражение ищет соответствие в рамках каждой из таких подстрок (листинг 30.40).

Листинг 30.40. Поиск соответствия в мультистроке. Файл modifier_without_m.rb

```
str = <<~here
      hello world
      ruby
      here

p /h[^\s]*o/.match str     #<MatchData "hello">
p /hello.*world/.match str #<MatchData "hello world">
p /hello.*ruby/.match str  # nil
```

Несмотря на то, что регулярное выражение `/hello.*ruby/` должно обнаруживать соответствие, этого не происходит, т. к. проверка регулярного выражения производится построчно.

При помощи модификатора `m` можно отключить такой режим поиска и считать весь мультистрочный текст одной строкой (листинг 30.41).

Листинг 30.41. Использование модификатора `m`. Файл modifier_m.rb

```
str = <<~here
      hello world
      ruby
      here

p /hello.*ruby/m.match str #<MatchData "hello world\nruby">
```

Модификатор `x` позволяет размещать внутри регулярного выражения комментарии. При этом пробельные символы игнорируются и позволяют добавлять в регулярное выражение форматирование (листинг 30.42).

Листинг 30.42. Использование модификатора `x`. Файл modifier_x.rb

```
str = <<~here
      hello world
      ruby
      here
```

```
regexp = /
  hello # Первое слово
  .*    # Любое количество символов
  ruby  # Второе слово
/mx

p regexp.match str #<MatchData "hello world\nruby">
```

Как видно из приведенного примера, допускается использовать одновременно несколько модификаторов.

Регулярные выражения допускают интерполяцию при помощи последовательности `#{}`. В листинге 30.43 в регулярное выражение подставляется результат вычисления `2 + 2`. В итоге получается выражение вида `/4/`.

Листинг 30.43. Интерполяция в регулярных выражениях. Файл interpolate.rb

```
p /#{2 + 2}/.match '2' # nil
p /#{2 + 2}/.match '4' #<MatchData "4">
```

При использовании модификатора `o` интерполируемое выражение вычисляется только один раз. В листинге 30.44 метод `number` возвращает либо `0`, либо `1` по случайному закону. В цикле проверяется соответствие регулярному выражению с модификатором и без модификатора `o`.

Листинг 30.44. Использование модификатора o. Файл modifier_o.rb

```
def number
  puts 'Создаем случайное число'
  rand(0..1)
end

puts 'Без модификатора o'
3.times do
  if /#{number}/.match '0'
    puts 'Значение 0'
  else
    puts 'Значение 1'
  end
end

puts 'С модификатором o'
3.times do
  if /#{number}/o.match '0'
    puts 'Значение 0'
  else
    puts 'Значение 1'
  end
end
```

Результатом выполнения программы будут следующие строки:

```
Без модификатора o
Создаем случайное число
Значение 0
Создаем случайное число
Значение 1
Создаем случайное число
Значение 1
С модификатором o
Создаем случайное число
Значение 0
Значение 0
Значение 0
```

Как можно видеть, без модификатора `o` метод `number` вызывается каждый раз, а с его использованием — лишь один раз.

30.7. Где использовать регулярные выражения?

Многие методы и конструкции в Ruby вместо строк могут принимать регулярные выражения. Их можно использовать в конструкции `case` (листинг 30.45).

Листинг 30.45. Использование регулярных выражений в `case`. Файл case.rb

```
case '500'
when /^\d+$/
  puts 'Число'
when /^(true|false)$/
  puts 'Логическое значение'
when String
  puts 'Строка'
else
  puts 'Неизвестный объект'
end
```

В `when`-условиях сравнение происходит с использованием оператора `===` (см. *разд. 8.5*), который перегружен для класса `Regexp`.

Методы `sub` и `gsub`, которые применяются для замены подстрок в строках, могут включать вместо подстрок регулярные выражения (листинг 30.46).

Листинг 30.46. Замена подстрок при помощи метода `sub`. Файл sub.rb

```
p 'Цена билета 500 рублей'.sub(/500/, '600') # "Цена билета 600 рублей"
```

Метод `sub` заменяет только первое вхождение, и если необходимо заменить несколько вхождений, можно воспользоваться методом `gsub` (листинг 30.47).

Листинг 30.47. Замена подстрок при помощи метода gsub. Файл gsub.rb

```
str = 'Цена билета увеличилась с 500 до 600 рублей'
p str.sub(/\d+/, '700') # "Цена билета увеличилась с 700 до 600 рублей"
p str.gsub(/\d+/, '700') # "Цена билета увеличилась с 700 до 700 рублей"
```

Методы `sub` и `gsub` могут принимать блок, в который передается найденная подстрока. В блоке можно принять решение, чем заменить текущую подстроку, и, вообще, следует ли осуществлять замену. В листинге 30.48 в строке `'Цена билета увеличилась с 500 до 600 рублей'` число `600` заменяется на `700`. Для этого в блоке указывается условие, позволяющее отсечь цифры, меньшие `600`.

Листинг 30.48. Использование блока в методе gsub. Файл gsub_block.rb

```
str = 'Цена билета увеличилась с 500 до 600 рублей'
str = str.gsub(/\d+/) { |x| x.to_i < 600 ? x : '700' }
puts str # Цена билета увеличилась с 500 до 700 рублей
```

Здесь метод `gsub` при помощи регулярного выражения `/\d+/` находит два соответствия: `'500'` и `'600'`. Они передаются внутрь блока в виде параметра `x`. Если значение `x` меньше `600`, оно остается неизменным, если больше или равно `600` — заменяется на подстроку `'700'`.

Методы `sub` и `gsub` можно использовать в том числе и для удаления подстрок — для этого в качестве второго аргумента следует указать пустую строку (листинг 30.49). Здесь из строки вырезаются все цифры.

Листинг 30.49. Удаление подстрок. Файл gsub_delete.rb

```
str = 'Цена билета увеличилась с 500 до 600 рублей'
puts str.gsub(/\d+/, '') # Цена билета увеличилась с  до  рублей
```

Метод `split`, который используется для разбивки строки в массив подстрок, в качестве разделителя может принять как строку, так и регулярное выражение. В листинге 30.50 строка разбивается по любому количеству пробельных символов.

Листинг 30.50. Разбивка строки по пробельным символам. Файл split.rb

```
str = <<~here
      hello    ruby

      world
      here

p str.split /\s+/ # ["hello", "ruby", "world"]
```

30.8. Примеры

Рассмотрим несколько примеров использования регулярных выражений. В листинге 30.51 из строки извлекается цена с плавающей точкой.

Листинг 30.51. Извлечение цены с плавающей точкой. Файл price_extract.rb

```
str = 'Цена билета 500.00 рублей'
result = str.match /\d+[.,]\d+/
puts result[0] # 500.00
```

Для извлечения числа `500.00` сначала срабатывает последовательность `\d+`, извлекающая цифры до точки. Далее используется диапазон `[.,]`, обозначающий один символ: либо точку, либо запятую. После этого опять указывается последовательность `\d+`, извлекающая произвольное количество оставшихся цифр.

Полученное регулярное выражение позволяет извлекать произвольную цену, будь то `5.00`, `50.00` или `5000.00` рублей:

```
 > 'Цена билета 5.00 рублей'.match /\d+[.,]\d+/
=> #<MatchData "5.00">
 > 'Цена билета 50.00 рублей'.match /\d+[.,]\d+/
=> #<MatchData "50.00">
 > 'Цена билета 5000.00 рублей'.match /\d+[.,]\d+/
=> #<MatchData "5000.00">
```

В листинге 30.52 представлено регулярное выражение для извлечения из строки даты.

Листинг 30.52. Извлечение даты из строки. Файл date_extract.rb

```
str = 'Дата представления 2019.10.28'
result = str.match /\d{4}\.\d{2}\.\d{2}/
puts result[0] # 2019.10.28
```

Дата имеет строго заданный формат. Так, годы всегда содержат четыре цифры — закодировать год можно при помощи последовательности `\d{4}`. Далее следует точка, для кодирования которой используется экранирование `\.` (если точку не экранировать, регулярные выражения будут искать любой символ). Затем идет месяц, в котором до октября одна цифра, а после октября — две. Поэтому для кодирования месяца используется последовательность `\d{1,2}`. Далее снова идет точка с экранированием и число, для кодирования которого можно применить последовательность `\d{1,2}`.

Для удобства можно поместить все компоненты в круглые скобки, чтобы извлекать их по отдельности. Более того, можно сразу преобразовать результирующий объект `MatchData` в массив (листинг 30.53).

Листинг 30.53. Файл date_extract_bracket.rb

```
str = 'Дата представления 2019.10.28'
result = str.match /(\d{4})\.(\d{2})\.(\d{2})/
p result      # 2019.10.28
p result.to_a # ["2019.10.28", "2019", "10", "28"]
```

Для более удобного доступа к результатам в круглых скобках им можно назначить говорящие имена (листинг 30.54).

Листинг 30.54. Файл date_extract_named.rb

```
str = 'Дата представления 2019.10.28'
result = str.match /(?<year>\d{4})\.(?<month>\d{2})\.(?<day>\d{2})/
p result      #<MatchData "2019.10.28" year:"2019" month:"10" day:"28">
puts result[:year]  # 2019
puts result[:month] # 10
puts result[:day]   # 28
```

Можно не только извлекать или удалять подстроки — допускается преобразовывать найденные значения. В листинге 30.55 приводится пример замены электронного адреса на тег `<a>` с атрибутом `href`. В браузере этот тег станет выглядеть как ссылка, переход по которой будет приводить к открытию окна создания письма в почтовой программе.

Листинг 30.55. Замена электронного адреса. Файл email.rb

```
str = 'Мой электронный адрес igorsimdyanov@gmail.com'
puts str.gsub(
  /[-0-9a-z_.]+@[-0-9a-z_.]+\.[a-z]{2,6}/,
  '<a href="mailto:\\0">\\0</a>'
)
```

Результатом выполнения программы будет следующая строка:

```
Мой электронный адрес <a href=\"mailto:igorsimdyanov@gmail.com\
                                                    ">igorsimdyanov@gmail.com</a>
```

Для имени в электронном адресе, которое располагается до символа `@`, можно использовать регулярное выражение `[-0-9a-z_.]+`. Символ дефиса выносится самым первым, чтобы он не воспринимался механизмом регулярных выражений как часть диапазона. Далее следует `@`, потом снова выражение `[-0-9a-z_.]+`, экранированная точка `\.` и выражение для домена первого уровня `[a-z]{2,6}`.

Для подстановки найденного результата в строку замены используется специальный синтаксис: `\\0`. Здесь число `0` соответствует индексу в результирующем `MatchData`-объекте. Если потребуется сослаться на первые фигурные скобки, можно использовать последовательность `\\1`, для вторых фигурных скобок — `\\2` и т. д (листинг 30.56).

Листинг 30.56. Файл email_alter.rb

```
str = 'Мой электронный адрес igorsimdyanov@gmail.com'
puts str.gsub(
  /([-0-9a-z_.]+)@[-0-9a-z_.]+\.[a-z]{2,6}/,
  '<a href="mailto:\\0">\\1</a>'
)
```

Результатом выполнения программы будет следующая строка:

```
Мой электронный адрес
<a href="mailto:igorsimdyanov@gmail.com">igorsimdyanov</a>
```

Пусть имеется текст с ценами в рублях и долларах — например: `'Цена билета в кино 700 рублей. Цена проезда $10'`. Необходимо рядом с цифрами, которым предшествует символ доллара, добавить в скобках перевод в рублях: `'Цена билета в кино 700 рублей. Цена проезда $10 (660 руб.)'`. Для этого можно воспользоваться опережающей проверкой `(?<=\$)` (листинг 30.57).

Листинг 30.57. Добавление перевода в рубли. Файл look_behind.rb

```
str = 'Цена билета в кино 700 рублей. Цена проезда $10'
puts str.gsub(/(?<=\$)\d+/) { |x| "#{x} (#{x.to_i * 66} руб.)" }
```

Результатом работы программы будет следующая строка

```
Цена билета в кино 700 рублей. Цена проезда $10 (660 руб.)
```

Пусть имеется текст, в котором нужно заменить все точки в конце предложения на троеточия. При этом важно не затронуть сокращения: `'г.'`, `'рис. '` и `'табл.'` — после преобразования они должны остаться неизменными:

```
На рис. 6 представлен график по данным табл. 8.
Как видно из рисунка, максимум приходится на 2002 г.,
поэтому целесообразно сосредоточиться на периоде
1998-2002 гг., как наиболее показательных.
```

Для решения этой задачи удобно воспользоваться негативной ретроспективной проверкой `(?<!...)`. Например, для замены всех точек на троеточие, не затрагивая последовательность `'г.'`, можно применить регулярное выражение `/(?<!г)\./` (листинг 30.58).

Листинг 30.58. Преобразование точек в троеточие. Файл look_behind_negative.rb

```
str = <<~here
        На рис. 6 представлен график по данным табл. 8.
        Как видно из рисунка, максимум приходится на 2002 г.,
        поэтому целесообразно сосредоточиться на периоде
        1998-2002 гг., как наиболее показательных.
      here
puts str.gsub(/(?<!г)\./, '...')
```

Результатом работы программы будут следующие строки:

```
На рис. .. 6 представлен график по данным табл... 8...
Как видно из рисунка, максимум приходится на 2002 г.,
поэтому целесообразно сосредоточиться на периоде
1998-2002 гг., как наиболее показательных...
```

Как видно из результата работы скрипта, все точки подвергаются замене, кроме тех, которым предшествует символ 'г'. Теперь остается обобщить решение на последовательности рис и табл. К сожалению, в скобках ретроспективных проверок не допускается использование метасимвола ИЛИ |. Объединение последовательностей "(?<!г)\.", "(?<!г)\." и "(?<!г)\." при помощи | также не проходит, поскольку проверка негативная, а не позитивная. Выйти из ситуации позволяет тот факт, что ретроспективные проверки допускают вложение. Поэтому три негативные ретроспективные проверки можно объединить в одну позитивную (листинг 30.59).

Листинг 30.59. Файл look_behind_final.rb

```
str = <<~here
        На рис. 6 представлен график по данным табл. 8.
        Как видно из рисунка, максимум приходится на 2002 г.,
        поэтому целесообразно сосредоточиться на периоде
        1998-2002 гг., как наиболее показательных.
      here
puts str.gsub(/(?<=((?<!г)(?<!рис)(?<!табл)))\./, '...')
```

Результатом работы программы будут следующие строки:

```
На рис. 6 представлен график по данным табл. 8...
Как видно из рисунка, максимум приходится на 2002 г.,
поэтому целесообразно сосредоточиться на периоде
1998-2002 гг., как наиболее показательных...
```

Задания

1. Создайте программу, которая, используя регулярные выражения, преобразует дату из формата 2003-03-21 в 21.03.2003.
2. Создайте программу, которая запрашивает у пользователя ввод числа в формате ###.##. Программа должна переспрашивать у пользователя ввод до тех пор, пока число не будет введено корректно.
3. Со страницы документации, посвященной классу String языка Ruby (например, **http://ruby-doc.org/core-2.6.1/String.html**), извлеките список методов. Сохраните полученный список в файл string.txt.
4. Установите набор программ ImageMagic для консольной обработки изображений. Воспользуйтесь утилитой identify для получения параметров изображения:

```
$ identify test.png
test.png PNG 1162x247 1162x247+0+0 8-bit sRGB 64763B 0.000u 0:00.000
```

Создайте Ruby-программу, которая при помощи утилиты `identify` будет получать информацию об изображении и выводить ширину и высоту изображения:

```
$ ruby identify.rb test.png
Ширина: 1162 px
Высота: 247 px
```

5. Решите предыдущую задачу по извлечению высоты и ширины изображения при помощи гема `RMagick` (**https://github.com/rmagick/rmagick**).
6. Установите утилиту `ffmpeg` для консольной обработки видеофайлов и потоков. Воспользуйтесь командой для получения информации о любом MP4-файле:

```
$ ffmpeg -i bf78c454b38d5.mp4
...
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'bf78c454b38d5.mp4':
  Metadata:
    major_brand     : isom
    minor_version   : 1
    compatible_brands: isomavc1mp42
    creation_time   : 2015-10-30T23:59:09.000000Z
  Duration: 00:03:48.37, start: 0.000000, bitrate: 7821 kb/s
    Stream #0:0(und): Video: h264 (Constrained Baseline) (avc1 /
              0x31637661), yuv420p(tv, bt709), 1920x1080 [SAR 1:1 DAR 16:9],
              7748 kb/s, 23.98 fps, 23.98 tbr, 24k tbn, 47.95 tbc (default)
    Metadata:
      creation_time   : 2015-10-30T23:58:17.000000Z
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo,
                                                    fltp, 71 kb/s (default)
    Metadata:
      creation_time   : 2015-10-30T23:58:17.000000Z
```

Создайте Ruby-программу, которая при помощи утилиты `ffmpeg` будет получать информацию об изображении и выводить ширину и высоту видеофайла:

```
$ ruby video.rb bf78c454b38d5.mp4
Ширина: 1920 px
Высота: 1080 px
```

7. Решите задачу по извлечению высоты и ширины изображения при помощи гема `streamio-ffmpeg` (**https://github.com/streamio/streamio-ffmpeg**).

ГЛАВА 31

Веб-программирование

Файлы с исходными кодами этой главы находятся в каталоге *rack* сопровождающего книгу электронного архива.

Веб-программирование, или создание веб-сайтов, является одной из главных ниш, которую занимает Ruby в сфере современных информационных технологий. В первую очередь это связано с популярностью флагманского веб-фреймворка Ruby-сообщества — Ruby on Rails (RoR). Этот фреймворк детально проработан и очень популярен, идеи и подходы, на которых он основан, часто вдохновляют разработчиков на других языках программирования для создания RoR-подобных фреймворков.

Ruby-разработчики часто начинают свой путь именно с Ruby on Rails, осваивая сам язык Ruby значительно позже или во время изучения Ruby on Rails. Этот подход работает, однако для RoR-разработчиков в коде Ruby может оставаться множество «магических» и непонятных мест.

Мы подойдем к проблеме веб-программирования со стороны гема `Rack`. Этот гем является краеугольным камнем при построении приложений на языке Ruby. Все веб-серверы, все Ruby-фреймворки используют этот гем как связующее звено.

В конце главы мы все же затронем Ruby on Rails. К сожалению, детальное описание самого фреймворка потребует объема, сопоставимого с объемом этой книги. Поэтому раздел, посвященный Ruby on Rails, можно рассматривать лишь как введение в фреймворк.

31.1. Протокол HTTP

Протокол HTTP используется для доставки веб-страниц и сопутствующих документов от веб-сервера к браузеру. Когда вы посещаете сайт и переходите по ссылкам, вы каждый раз шлете HTTP-запрос серверу. В ответ сервер присылает HTTP-ответ, чаще всего — HTML-страницу. Эта страница содержит структурированный текст и ссылки на изображения, аудио- и видеоматериалы (рис. 31.1).

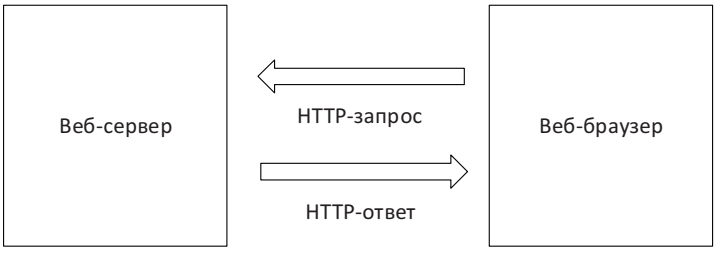


Рис. 31.1. Протокол HTTP имеет клиент-серверную природу

Чтобы браузер имел возможность отрисовать страницу, он должен загрузить с сервера HTML-страницу и все сопутствующие материалы. Для обслуживания таких запросов и предназначен протокол HTTP.

Поскольку с одной стороны находится клиент (браузер), а с другой стороны — сервер, протокол называют *клиент-серверным*. Клиенты отправляют HTTP-запросы, чтобы получить какой-либо ресурс. Веб-сервер обрабатывает запрос и отправляет запрашиваемый ресурс в виде HTTP-ответа. Клиент и сервер обмениваются друг с другом HTTP-документами. Документ делится на две части: HTTP-заголовки и тело документа (рис. 31.2).

Рис. 31.2. HTTP-документ

HTTP-заголовки содержат различную метаинформацию, в то время как в теле документа передается полезная нагрузка — например, HTML-страница или изображение.

Посмотреть содержимое HTTP-документа проще всего, открыв в браузере панель веб-разработчика (рис. 31.3).

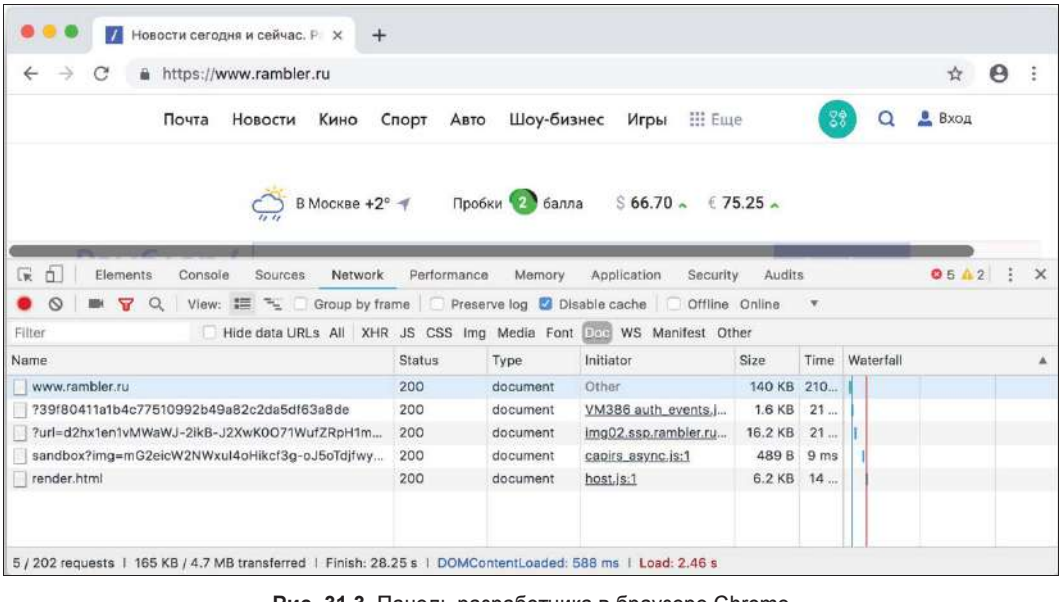


Рис. 31.3. Панель разработчика в браузере Chrome

В панели отображаются HTTP-запросы, которые были отправлены серверу. Если выбрать один из HTTP-запросов, можно увидеть HTTP-заголовки запроса и ответа (рис. 31.4).

На вкладке **Response** (Ответ) можно посмотреть содержимое HTTP-документа. Так, на рис. 31.5 в качестве ответа представлена HTML-страница.

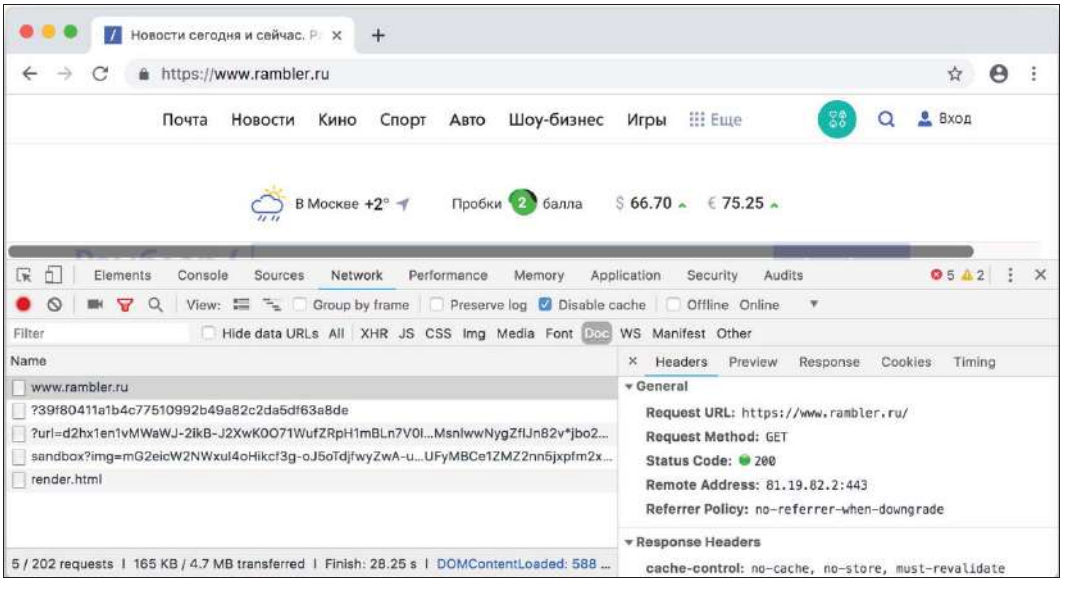


Рис. 31.4. HTTP-заголовки

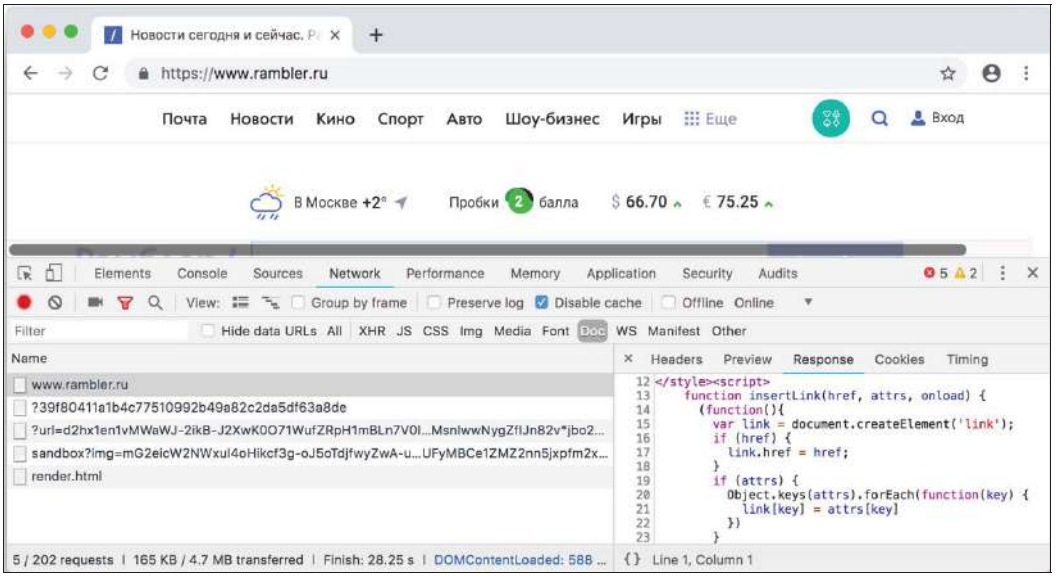


Рис. 31.5. Тело HTTP-запроса

31.1.1. HTTP-заголовки

Как видно из рис. 31.4, HTTP-заголовков очень много — как со стороны клиента, так и со стороны сервера. Разобраться со всеми HTTP-заголовками мы здесь не сможем, поскольку полное изложение протокола HTTP выходит за рамки этой книги. Но несколько наиболее типичных HTTP-заголовков мы все же рассмотрим:

- `GET /path HTTP/1.1` — при запросе клиент всегда отправляет метод, с помощью которого осуществляется запрос. В нашем случае это метод `GET`, при котором отправляются только заголовки, тело HTTP-документа остается пустым. Если клиент захочет отправить какой-либо файл, скорее всего, для этого будет использоваться метод `POST`. После имени метода следует путь от корня сайта `/path` до страницы, к которой обращается клиент. Далее следует версия протокола: `HTTP 1.1`. В настоящий момент протокол HTTP версии 1.0 практически не используется, основная масса запросов основана на версии 1.1, однако появляется поддержка протокола 2.0;
- `Host: rambler.ru` — каждый клиент обязательно отправляет HTTP-заголовок `Host`, в котором передается доменное имя сайта. Если сервер обслуживает несколько сайтов, по этому заголовку он будет определять, к какому из сайтов адресован запрос;
- `Accept: text/html` — сообщает, что клиент предпочитает получить заголовки в каком-либо определенном формате, в нашем случае — в виде HTML-страницы. Если у сервера есть несколько вариантов ответа на текущий запрос, он выдаст наиболее подходящий. Если вариант только один, сервер выдаст то, что есть.

Мы рассмотрели несколько заголовков, которые посылает браузер серверу. В ответ сервер, помимо HTTP-документа, тоже шлет заголовки:

- `HTTP/1.1 200 OK` — сообщает о статусе ответа. Заголовок начинается с версии протокола HTTP. Далее следует код ответа — в нашем случае `200` (успешная обработка запроса);
- `Content-Type: text/html` — сообщает о формате присланного документа, в нашем случае это HTML-страница;
- `Server: nginx` — информационный заголовок, сообщающий название веб-сервера, обслуживающего запрос.

Это лишь часть возможных HTTP-заголовков — их, как правило, гораздо больше.

31.1.2. HTTP-коды ответа

Каждый ответ сервера сопровождается HTTP-кодом ответа. Они всегда трехзначные и начинаются с одной из цифр: от `1` до `5` (табл. 31.1).

В предыдущем разделе мы познакомились только с одним из них: `200` — код успешно обработанного запроса. Однако можно получить и другие коды — к примеру, `201`, если в результате запроса что-то создается (например, комментарий или статья). Код `201` как раз и сообщает об успешном создании запрошенного ресурса.

Таблица 31.1. *HTTP-коды ответа*

HTTP-коды	Описание
1xx	Информационные коды, сервер пребывает в процессе обработки запроса
2xx	Коды успешного выполнения запроса
3xx	Коды переадресации
4xx	Коды ошибочного запроса со стороны клиента
5xx	Коды ошибок на стороне сервера

Коды, начинающиеся с цифры 3, обозначают различного рода переадресации. Как правило, вместе с таким HTTP-кодом приходит заголовок Location, в котором указывает адрес, по которому следует перейти браузеру, чтобы получить запрашиваемый ресурс. HTTP-код 301 сообщает о том, что переадресация постоянная, а код 302 — что временная.

Коды, которые начинаются с цифры 4, сообщают о неверном запросе со стороны клиента. Страница может просто отсутствовать — в этом случае сервер возвращает код 404. К странице может быть закрыт доступ — в этом случае сервер вернет код 403. Если запрос со стороны клиента не корректный (например, он слишком длинный) — можно получить статус 400.

Коды, начинающиеся с цифры 5, обозначают ошибку на стороне сервера — либо в его конфигурации, либо в коде. Большинство ошибок и исключений, которые станут возникать в вашей программе, будут видны пользователям как ответ 500 со стороны сервера.

31.2. Веб-серверы

Для того чтобы создать свой сайт, не нужно разрабатывать собственный сервер и реализовывать протокол HTTP. Существует немало уже готовых веб-серверов. Эти серверы вы и будете встраивать в свое приложение — зачастую в виде гема (рис. 31.6).

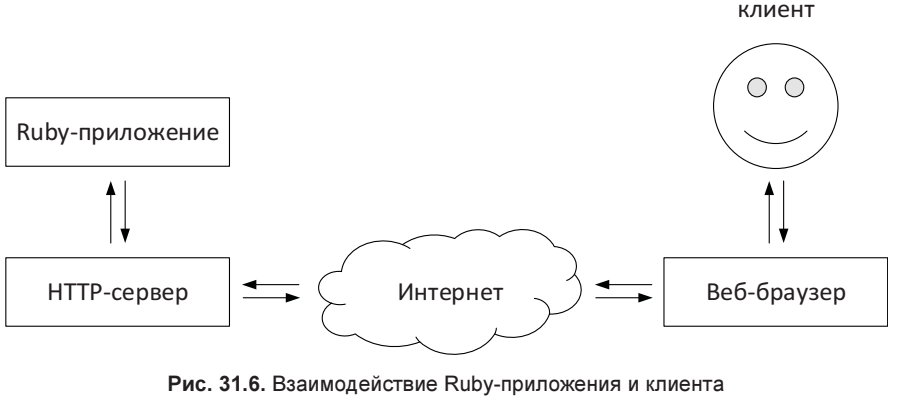


Рис. 31.6. Взаимодействие Ruby-приложения и клиента

Рассмотрим кратко некоторые из них:

- WebBrick — самый простой сервер, он уже встроен в Ruby. Его можно использовать, не устанавливая дополнительных библиотек и гемов. Этот сервер предназначен только для локальной разработки, он однопоточный и не может обслуживать сразу множество клиентов;
- Thin — это также однопоточный сервер, который реализует механизм EventLoop: один поток по очереди опрашивает все соединения в неблокирующем режиме. Если ответ полностью получен, веб-сервер начинает его обработку, если ответ не получен, он переключается на следующее соединение. Таким образом, один поток может эффективно обрабатывать параллельные запросы от множества клиентов. Преимущество этого подхода заключается в том, что процессору нет необходимости переключать контекст с одного параллельного процесса на другой, — в результате при массовых запросах достигается экономия процессорного времени. У Thin имеется недостаток — он «боится» медленной обработки запросов со стороны Ruby-приложения. Если приложение «задумается», EventLoop зависнет — поток не сможет бежать дальше по кругу и опрашивать остальные соединения. Поэтому в продакшен-среде чаще всего используют описанные далее серверы Unicorn и Puma;
- Unicorn — это классический fork-сервер. Для параллельного обслуживания запросов в нем создается несколько процессов, которые называют *воркерами* (worker). Каждый процесс может одновременно обслуживать лишь одно соединение, однако за счет того, что их несколько, решается проблема с параллельным обслуживанием запросов. Однако Unicorn «боится» медленных клиентов. Поскольку этот сервер организован в виде отдельных процессов, он потребляет весьма много оперативной памяти. Из-за этого удается запустить лишь ограниченное количество воркеров. Если к серверу присоединится много клиентов с медленными каналами связи, они могут занять все воркеры, и их не останется для обслуживания других клиентов. Поэтому Unicorn часто используется в связке с другими веб-серверами — например, с nginx. Такой дополнительный сервер не в состоянии обслуживать Ruby-код, зато он может обрабатывать статические запросы (изображения, CSS/JS-файлы и т. п.);
- Puma — комбинирует подходы, принятые в Unicorn и Thin. Сервер поддерживает EventLoop, кроме того, имеется возможность запускать несколько воркеров. Поэтому в настоящий момент Puma — это наиболее популярный веб-сервер, который часто используется в продакшен-окружении.

31.3. Гем *Rack*

Итак, мы посмотрели на работу веб-клиента и браузера, а также познакомились с основными веб-серверами. Теперь мы разберемся, как в обработку запросов от клиентов встраивается гем `Rack`, как запросы от клиентов попадают в Ruby-код, а ответ Ruby-программы отправляется в браузер.

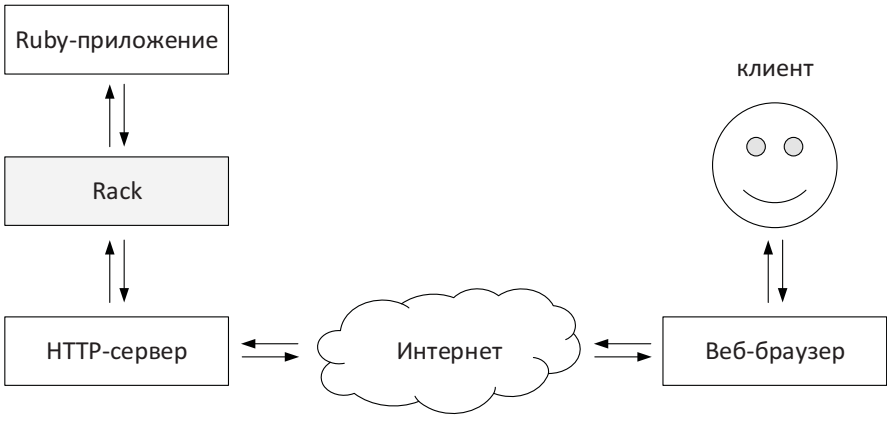


Рис. 31.7. Гем `Rack`

Запросы, которые отправляются клиентами с помощью веб-браузера, попадают через Интернет к веб-серверу (рис. 31.7).

Далее веб-сервер должен обратиться к Ruby-приложению. Как мы видели ранее, веб-серверов достаточно много, да и Ruby-приложения также разрабатываются с участием разных фреймворков. Для того, чтобы связать веб-серверы и Ruby-приложения, как раз и предназначен гем `Rack`. Он выступает в качестве единого интерфейса для взаимодействия Ruby-приложения и веб-серверов. Все современные Ruby-фреймворки взаимодействуют с серверами через `Rack`.

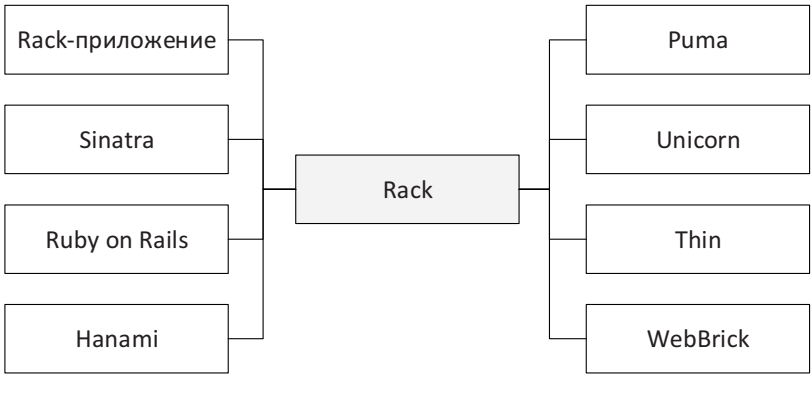


Рис. 31.8. `Rack` — единая спецификация для веб-фреймворков и веб-серверов

На рис. 31.8 *слева* представлены различные варианты веб-приложений, созданные на Ruby. Это может быть и «чистое» Rack-приложение, и какой-либо Ruby-фреймворк — например, Sinatra, Ruby on Rails или Hanami. *Справа* на рис. 31.8 представлены различные веб-серверы. Фреймворки и веб-серверы для связи друг с другом используют один и тот же гем `Rack`. Он выступает здесь в роли спецификации, определяя, как веб-сервер взаимодействует с Ruby-приложением.

31.3.1. Установка гема *Rack*

Для установки гема можно воспользоваться утилитой `gem`, которая детально рассматривалась в *разд. 3.6*:

```
$ gem install rack
Fetching rack-2.0.6.gem
Successfully installed rack-2.0.6
Parsing documentation for rack-2.0.6
Installing ri documentation for rack-2.0.6
Done installing documentation for rack after 2 seconds
1 gem installed
```

Гем `Rack` предоставляет утилиту `rackup`, при помощи которой можно запускать веб-сервер. Убедиться в том, что утилита успешно установлена, можно, запросив текущую версию гема при помощи параметра `-v`:

```
$ rackup -v
Rack 1.3 (Release: 2.0.6)
```

31.3.2. Простейшее *Rack*-приложение

Если запустить утилиту `rackup` без параметров, по умолчанию она ищет конфигурационный файл config.ru, в котором должно быть определено `Rack`-приложение.

Для того чтобы воспользоваться библиотекой `Rack`, ее необходимо подключить при помощи метода `require`. `Rack`-приложение представляет собой объект, который отзывается на метод `call`. Метод `call` должен принимать единственный параметр, через который внутрь метода передаются переменные окружения, информация от клиента и HTTP-заголовки.

В листинге 31.1 представлено `Rack`-приложение на базе класса `App`. Метод `call` реализован в виде синглетон-метода.

Листинг 31.1. Конфигурационный файл config.ru. Файл app/config.ru

```
require 'rack'

class App
  def self.call(env)
    [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
  end
end

run App
```

В качестве ответа метод `call` должен возвращать массив с тремя элементами:

- HTTP-код ответа — например, `200`;
- хэш с HTTP-заголовками ответа: ключом выступает название HTTP-заголовка, значением — содержимое;
- массив с содержимым тела HTTP-документа.

Чтобы запустить приложение, методу `run` в качестве аргумента необходимо передать класс `App`.

Для запуска веб-сервера необходимо выполнить команду `rackup` без параметров:

```
$ rackup
[2019-02-03 17:40:56] INFO  WEBrick 1.4.2
[2019-02-03 17:40:56] INFO  ruby 2.6.0 (2018-12-25) [x86_64-darwin15]
[2019-02-03 17:40:56] INFO  WEBrick::HTTPServer#start: pid=7073 port=9292
```

Утилита сообщает справочную информацию: название веб-сервера (в нашем случае — `WebBrick`), версию Ruby-интерпретатора и порт, на котором запущен сервер. Пока программа работает, веб-сервер ожидает запросы на порту `9292`. Остановить сервер можно при помощи комбинации клавиш <Ctrl>+<C>.

В сети каждый хост имеет IP-адрес, по которому один компьютер может отличаться от другого. Однако на хосте могут работать несколько серверов. Чтобы их отличать друг от друга, используется *порт* — специальный уникальный идентификатор, который указывается при установке сетевого соединения с хостом.

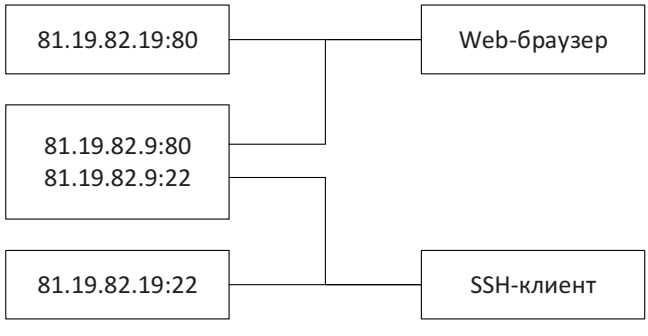


Рис. 31.9. Порты позволяют размещать на одном хосте несколько серверов

Некоторые порты стандартизированы — например, порт `80` используется для HTTP-сервера, а `22` — для SSH-сервера (рис. 31.9). Порты до `1024` считаются привилегированными — для них не всегда можно запустить сервер, если нет соответствующих прав доступа (как правило, требуются права суперпользователя). Из-за этого при разработке популярны порты со значениями выше `1024`. Именно поэтому по умолчанию утилита `rackup` и использует порт `9292`.

Если теперь обратиться в браузере по адресу **http://localhost:9292/**, можно увидеть ответ, сформированный `Rack`-приложением (рис. 31.10).

Псевдоним `localhost` — это синоним для IP-адреса 127.0.0.1, который зарезервирован за локальным компьютером. Все запросы, отправляемые на этот адрес, попадают на текущий хост, минуя любые другие сетевые узлы.

Как видно из рис. 31.10, получено приветствие **Hello, world!**, которое было отправлено из `Rack`-приложения (листинг 31.1). На вкладке **Network** (Сеть) панели **Инструменты разработчика** показано единственное обращение — с кодом ответа **200**.

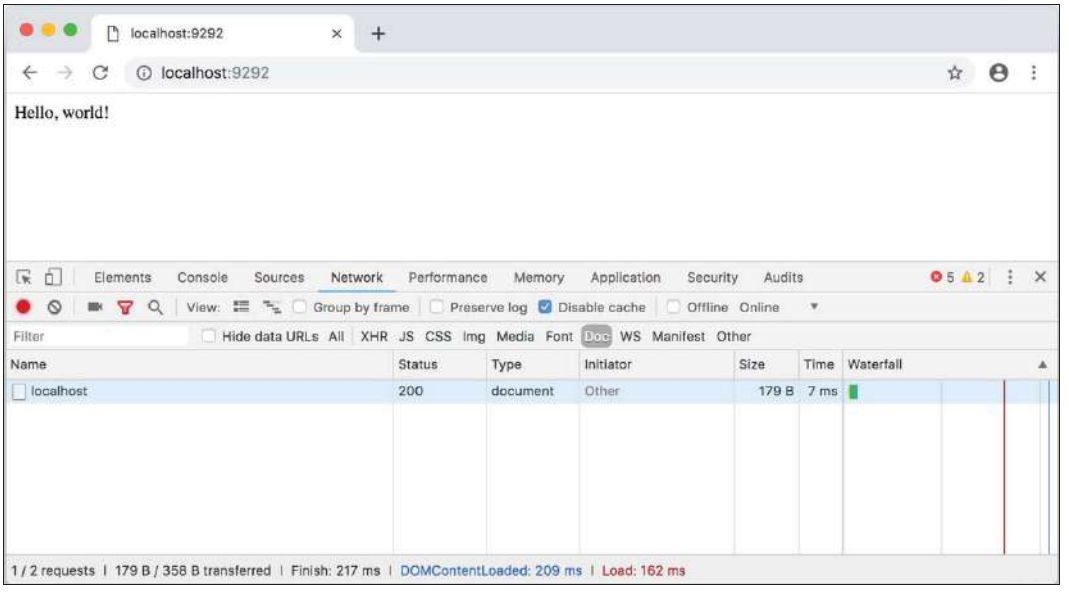


Рис. 31.10. Ответ `Rack`-приложения

Среди HTTP-заголовков ответа можно обнаружить следующие:

- `Content-Type: text/plain`
- `Date: Sun, 03 Feb 2019 15:04:54 GMT`
- `Server: WEBrick/1.4.2 (Ruby/2.6.0/2018-12-25)`

HTTP-заголовок `Content-Type` мы задали самостоятельно при формировании метода `call`. Остальные HTTP-заголовки: дата формирования ответа `Date` и информационный заголовок `Server` — были автоматически сформированы гемом `Rack`.

Если несколько раз перезагрузить страницу и обратиться к консоли сервера, можно обнаружить журнальные сообщения о каждом из запросов:

```
127.0.0.1 - - [03/Feb/2019:19:33:15 +0300] "GET / HTTP/1.1" 200 - 0.0007
127.0.0.1 - - [03/Feb/2019:19:33:15 +0300] "GET /favicon.ico HTTP/1.1" 200 -
0.0005
127.0.0.1 - - [03/Feb/2019:19:33:17 +0300] "GET / HTTP/1.1" 200 - 0.0046
127.0.0.1 - - [03/Feb/2019:19:33:17 +0300] "GET /favicon.ico HTTP/1.1" 200 -
0.0004
127.0.0.1 - - [03/Feb/2019:19:33:18 +0300] "GET / HTTP/1.1" 200 - 0.0003
127.0.0.1 - - [03/Feb/2019:19:33:18 +0300] "GET /favicon.ico HTTP/1.1" 200 -
0.0003
127.0.0.1 - - [03/Feb/2019:19:33:19 +0300] "GET / HTTP/1.1" 200 - 0.0003
127.0.0.1 - - [03/Feb/2019:19:33:19 +0300] "GET /favicon.ico HTTP/1.1" 200 -
0.0005
```

Помимо запроса к главной странице /, браузер каждый раз посылает запрос для значка `favicon.ico`, который используется для отрисовки логотипа сайта на вкладке.

В листинге 31.2 представлен исправленный вариант `Rack`-приложения, в котором содержимое заголовка `Content-Type` поправлено на `text/html`, а содержимое оформлено в виде HTML-страницы с заголовком 1-го уровня `'Hello, Ruby!'`, отформатированным тегом `<h1>`.

Листинг 31.2. Отправка HTML-страницы. Файл html/config.ru

```
require 'rack'

class App
  class << self
    def call(env)
      [200, { 'Content-Type' => 'text/html' }, [template('Hello, Ruby!')]]
    end

    def template(name)
      <<~HTML
        <!DOCTYPE html>
        <html lang="ru">
        <head>
          <title>#{name}</title>
          <meta charset='utf-8'>
        </head>
        <body>
          <h1>#{name}<h1>
        </body>
        </html>
      HTML
    end
  end
end

run App
```

Для того чтобы изменения вступили в силу, потребуется остановить веб-сервер при помощи комбинации клавиш <Ctrl>+<C> и запустить его снова. Ответ `Rack`-приложения представлен на рис. 31.11.

Как можно видеть, текст **Hello, world!** был заменен на **Hello, Ruby!**. Если мы посмотрим на необработанный ответ веб-сервера, то сможем обнаружить следующий HTML-код:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Hello, Ruby!</title>
  <meta charset='utf-8'>
</head>
```

```
<body>
  <h1>Hello, Ruby!</h1>
</body>
</html>
```

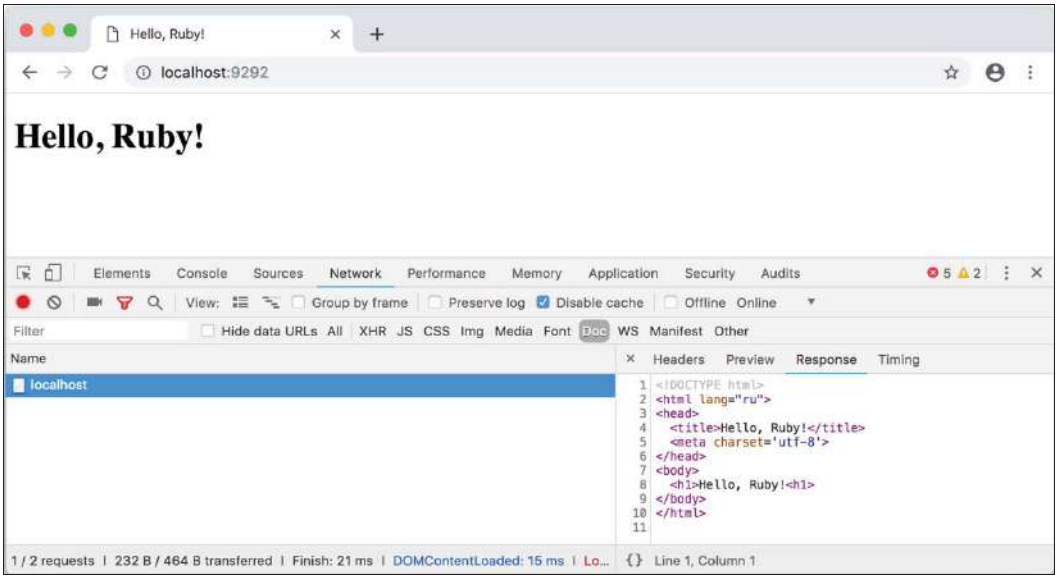


Рис. 31.11. Ответ `Rack`-приложения

31.3.3. Управление утилитой *rackup*

При запуске утилиты `rackup` можно явно указывать название конфигурационного файла:

```
$ rackup config.ru
[2019-02-03 19:43:58] INFO  WEBrick 1.4.2
[2019-02-03 19:43:58] INFO  ruby 2.6.0 (2018-12-25) [x86_64-darwin15]
[2019-02-03 19:43:58] INFO  WEBrick::HTTPServer#start: pid=11492 port=9292
```

Поэтому, если `Rack`-приложение будет размещено в каком-либо другом файле, его можно явно указать первым параметром утилиты `rackup`.

Если текущий порт `9292` не удобен или занят каким-либо другим сервером, можно задать произвольный порт при помощи параметра `-p`:

```
$ rackup -p 8080
[2019-02-03 19:47:33] INFO  WEBrick 1.4.2
[2019-02-03 19:47:33] INFO  ruby 2.6.0 (2018-12-25) [x86_64-darwin15]
[2019-02-03 19:47:33] INFO  WEBrick::HTTPServer#start: pid=11649 port=8080
```

31.3.4. Обработка несуществующих страниц

Rack-приложение из листинга 31.1 на любой запрос отправляет один и тот же ответ. Не имеет значения, какой запрос вбивается в адресную строку:

```
http://localhost:9292/
http://localhost:9292/hello
```

Все они выдают один результат: код ответа 200 и HTML-сообщение: 'Hello, world!'.

В листинге 31.3 представлен вариант Rack-приложения, который выдает код 200 для главной страницы **http://localhost:9292/**, а для всех остальных возвращается 404 — страница не существует. Для этого потребуется обратиться к параметру env и извлечь из него путь к текущей странице env['PATH_INFO'].

Листинг 31.3. Обработка несуществующих страниц. Файл 404/config.ru

```
require 'rack'

class App
  def self.call(env)
    headers = { 'Content-Type' => 'text/plain' }
    code = 200
    body = 'Hello, world!'

    unless env['PATH_INFO'] == '/'
      code = 404
      body = 'Not Found'
    end

    [code, headers, [body]]
  end
end

run App
```

По умолчанию приложение возвращает код ответа 200, а сообщение — 'Hello, world!'. Однако, если путь env['PATH_INFO'] не совпадает с главной страницей, меняем код на 404, а тело ответа на 'Not Found'. На последней строке метода call формируется массив из трех элементов, который должно возвращать любое Rack-приложение.

Теперь при обращении к несуществующему адресу веб-сервер будет возвращать код **404** (рис. 31.12).

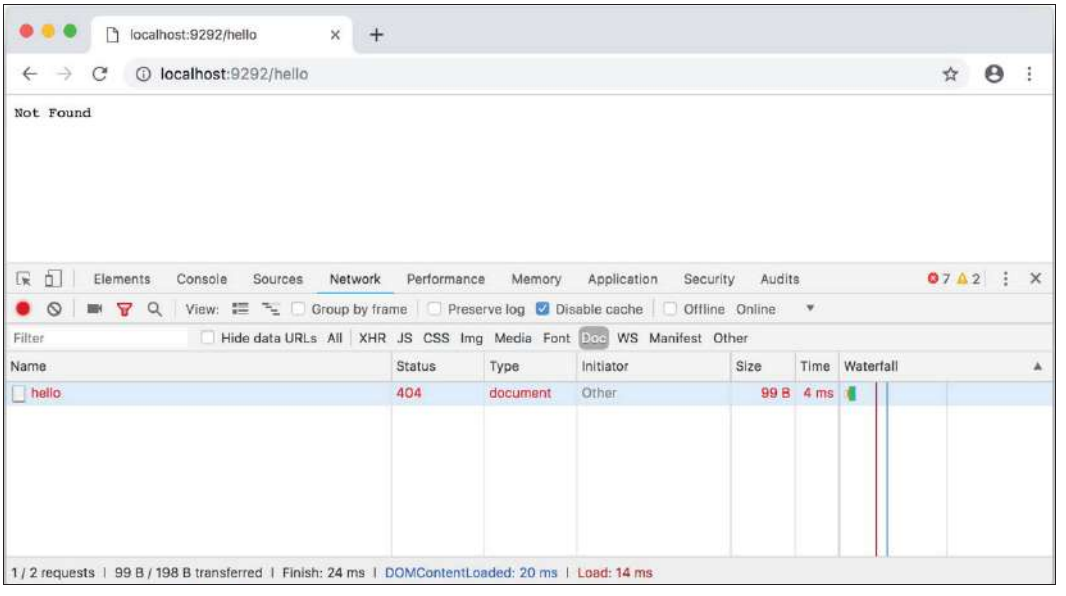


Рис. 31.12. Возврат статуса **404** при обращении к несуществующей странице

31.3.5. Размер HTTP-содержимого

Научим `Rack`-приложение передавать размер HTTP-документа, чтобы клиент мог оценивать объем загружаемой страницы. В случае небольших HTML-документов отсутствие этой информации не критично. Однако, если в ответ на запрос отправляется объемный файл, браузер не сможет сориентировать пользователя, когда завершится загрузка. Более того, браузер не сможет загружать содержимое файла параллельно в несколько потоков.

Для передачи размера HTTP-документа в протоколе HTTP предусмотрен заголовок `Content-Length` — в качестве значения в нем указывается размер в байтах. В листинге 31.4 в хэш `headers` добавляется еще один ключ: `'Content-Length'`. Значение для этого ключа вычисляется по размеру строки `body`. Причем преобразуем мы полученное значение к строковому значению при помощи метода `to_s` — `Rack` будет ожидать именно строку.

Листинг 31.4. Размер HTTP-документа. Файл content_length/config.ru

```
require 'rack'

class App
  def self.call(env)
    headers = { 'Content-Type' => 'text/plain' }
    code = 200
    body = 'Hello, world!'
```

```
    unless env['PATH_INFO'] == '/'
      code = 404
      body = 'Not Found'
    end

    headers['Content-Length'] = body.length.to_s

    [code, headers, [body]]
  end
end

run App
```

Если теперь перезапустить сервер и обратиться к браузеру, в детальном отчете о запросе к серверу можно обнаружить HTTP-заголовок `Content-Length` (рис. 31.13).

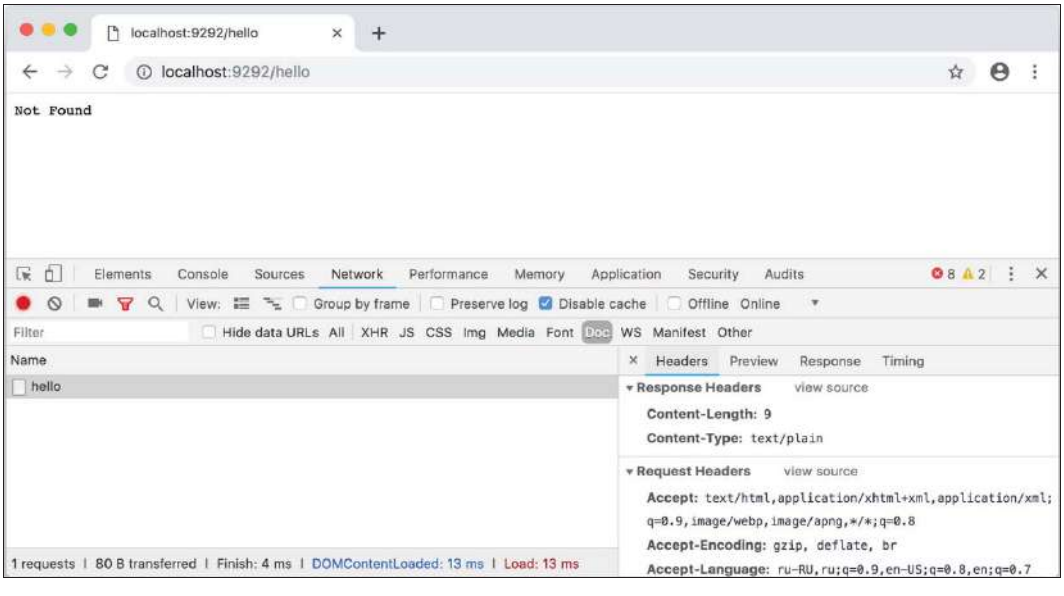


Рис. 31.13. Возврат HTTP-заголовка `Content-Length`

Как можно видеть, для несуществующей страницы HTTP-заголовок `Content-Length` установлен в значение **9**. При этом полезная нагрузка ответа составляет строку **Not Found**, которая как раз и состоит из 9 символов.

В случае, если тело HTML-документа содержит русский текст или бинарную информацию, вместо метода `length` лучше воспользоваться методом `bytesize`, который возвращает размер строки в байтах, а не в символах:

```
 > 'Hello, world!'.length
=> 13
 > 'Hello, world!'.bytesize
=> 13
```

```
 > 'Привет, мир!'.length
=> 12
 > 'Привет, мир!'.bytesize
=> 21
```

31.3.6. *Proc*-объект в качестве *Rack*-приложения

Метод `run` может принимать не только объект, который отзывается на метод `call`, но и обычный `proc`-объект. В листинге 31.5 представлено минимально возможное `Rack`-приложение.

Листинг 31.5. Использование `proc`-объекта. Файл proc/config.ru

```
require 'rack'
run proc { |env|
  [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
}
```

Вместо `Proc`-объекта можно использовать `lambda` — такое `Rack`-приложение работает аналогичным образом.

Листинг 31.6. Использование `lambda`-объекта. Файл lambda/config.ru

```
require 'rack'

run ->(env) do
  [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
end
```

31.3.7. Промежуточные слои (middleware)

Расширять приложение допускается при помощи *промежуточных слоев* (middleware), которые можно выстраивать в цепочки вызовов. Такие промежуточные слои удобно создавать в виде классов, которые тоже должны реализовывать метод `call` (листинг 31.7).

Листинг 31.7. Использование промежуточных слоев. Файл middleware/config.ru

```
require 'rack'

class MyMiddleware
  def initialize(app)
    @app = app
  end

  def call(env)
    unless env['PATH_INFO'] == '/'
```

```
      return [404, { 'Content-Type' => 'text/plain' }, ['Not Found']]
    end
    @app.call(env)
  end
end

use MyMiddleware

run ->(env) do
  [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
end
```

Здесь методу `MyMiddleware#initialize` передается объект `Rack`-приложения, который сохраняется в инстанс-переменной `@app`. Эта инстанс-переменная используется в методе `call`. Внутри метода `call` можно реализовать бизнес-логику — например, возвращать HTTP-код `404`, если происходит обращение к любой странице, кроме главной.

Для того чтобы воспользоваться промежуточным слоем `MyMiddleware`, его необходимо передать методу `use`.

31.3.8. Роутинг

Гем `Rack` предоставляет метод `map`, который назначает обработчики роутам. Метод принимает в качестве первого параметра строку с роутом, в блоке метода задается обработчик.

Гем `Rack` содержит в примерах небольшое приложение-пасхалку, которое при помощи псевдографики рисует в браузере изображение лобстера. Подключить его можно при помощи инструкции `require`. В листинге 31.8 приложение запускается по адресу **http://localhost:9292/lobster**.

Листинг 31.8. Использование промежуточных слоев. Файл lobster/config.ru

```
require 'rack'
require 'rack/lobster'

map '/lobster' do
  run Rack::Lobster.new
end

run ->(env) do
  [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
end
```

После перезапуска веб-сервера обращение к адресу **http://localhost:9292/lobster** будет приводить на страницу с изображением лобстера (рис. 31.14).

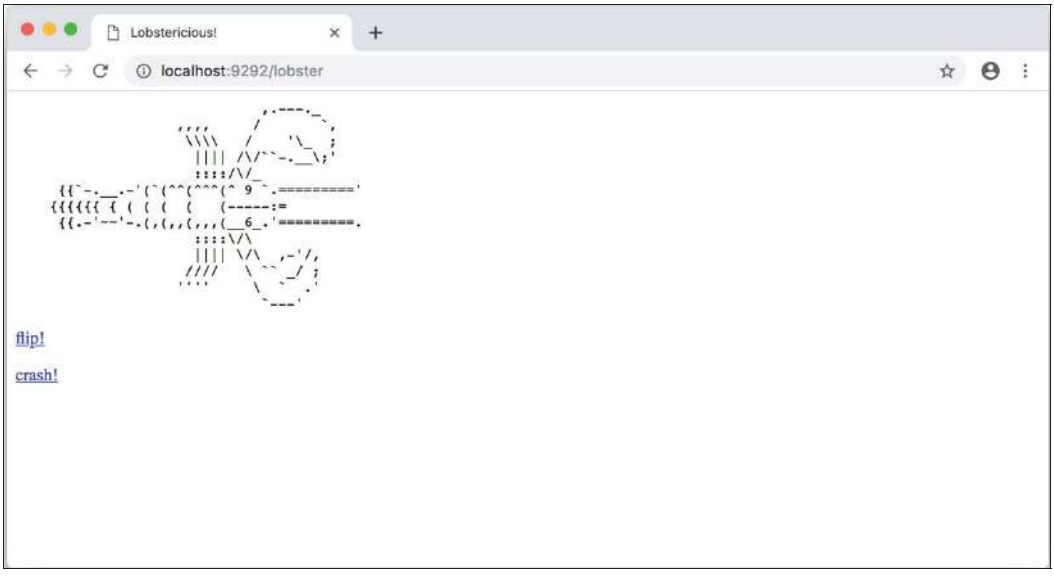


Рис. 31.14. Результат работы lobster-приложения

Нажатие на ссылку **flip!** приведет к переворачиванию изображения, повторное нажатие на нее вернет лобстера в исходное положение.

31.3.9. Обработка статических файлов

Статическими файлами называют такие файлы, для которых не требуется дополнительной обработки на стороне сервера. К ним относят изображения, CSS-файлы, JS-скрипты. Ruby-файлы относят к *динамическим файлам*, т. к. перед отправкой результата клиенту необходимо их выполнить, получить вывод в стандартный поток, оформить его в виде HTTP-документа и лишь затем отправить полученный документ.

Разработаем `Rack`-приложение, которое сможет отдавать любые статические файлы просто по обращению к ним через адресную строку. Для этого в каталоге static/images разместим файл ruby.jpg с логотипом языка Ruby (рис. 31.15). Приложение будет отдавать этот файл по адресу **http://localhost:9292/images/ruby.jpg**.

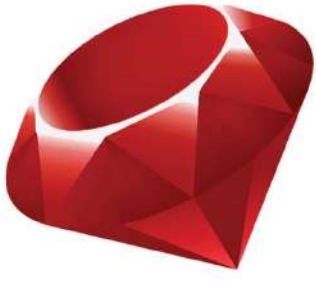


Рис. 31.15. Логотип языка Ruby

В `Rack`-приложении при помощи метода `map` создадим новый роут `'/images'`. В блоке метода `map` запустим `Rack`-приложение `Rack::File`, передав ему в качестве аргумента каталог `'images'` (листинг 31.9).

Листинг 31.9. Обработка статических файлов. Файл static/config.ru

```
require 'rack'

map '/images' do
  run Rack::File.new 'images'
end

run ->(env) do
  [200, { 'Content-Type' => 'text/plain' }, ['Hello, world!']]
end
```

Теперь, если обратиться по адресу **http://localhost:9292/images/ruby.jpg**, можно получить логотип языка Ruby.

Для того чтобы встроить изображение в HTML-страницу, потребуется сослаться на нее при помощи HTML-тэга `<img>`. В листинге 31.10 представлено `Rack`-приложение, которое выводит изображение и взятый из Википедии текст с описанием языка Ruby.

Листинг 31.10. Обработка статических файлов. Файл static_html/config.ru

```
require 'rack'

map '/images' do
  run Rack::File.new 'images'
end

class App
  attr_accessor :title, :description

  def initialize(title: title, description: description)
    @title = title
    @description = description
  end

  def call(env)
    [200, { 'Content-Type' => 'text/html' }, [template]]
  end

  def template
    format(HTML, title: title, description: description)
  end
```

```
  HTML = <<~HTML
           <!DOCTYPE html>
           <html lang="ru">
           <head>
             <title>%<title>s</title>
             <meta charset='utf-8'>
           </head>
           <body>
             <h1>%<title>s</h1>
             <p>
               <img
                 src='/images/ruby.jpg'
                 with='100'
                 height='100'
                 style='float: left; padding: 0 10px 10px 0' />
               %<description>s
             </p>
           </body>
           </html>
         HTML
end

run App.new(
  title: 'Язык программирования Ruby',
  description: <<~RUBY
    Ruby — динамический, рефлективный, интерпретируемый высокоуровневый
    язык программирования. Язык обладает независимой от операционной
    системы реализацией многопоточности, сильной динамической типизацией,
    сборщиком мусора и многими другими возможностями. По особенностям
    синтаксиса он близок к языкам Perl и Eiffel,
    по объектно-ориентированному подходу — к Smalltalk.
  RUBY
)
```

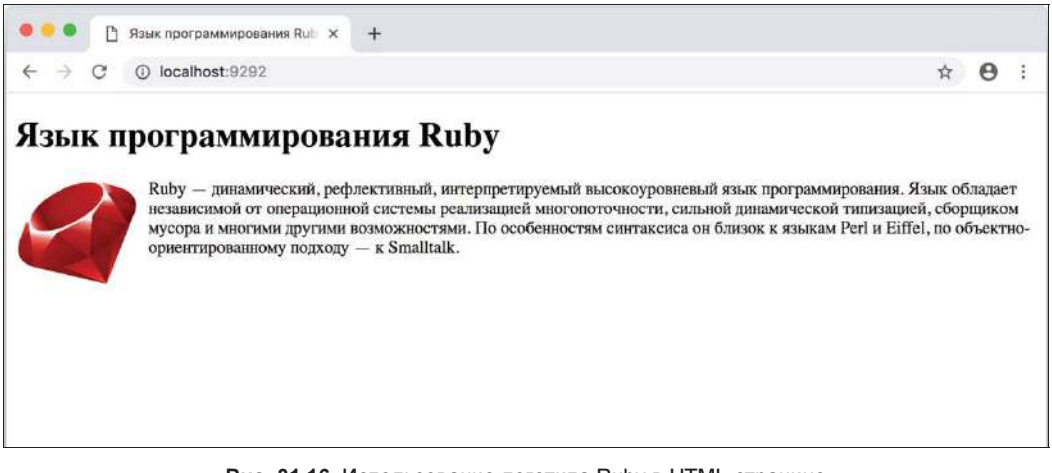


Рис. 31.16. Использование логотипа Ruby в HTML-странице

В приведенном примере вместо реализации синглетон-метода `call` реализован одноименный инстанс-метод. Это позволяет создать объект класса `App` и передать название страницы и ее текстовое содержимое в виде параметров метода `new`. Результат работы программы из листинга 31.10 представлен на рис. 31.16.

31.4. Ruby on Rails

Ruby on Rails — это фреймворк, предназначенный для быстрой разработки веб-сайтов. Здесь мы сможем дать лишь самое краткое введение в фреймворк, для его полного освещения потребуется объем, сопоставимый с объемом всей этой книги.

31.4.1. Введение в Ruby on Rails

Популярность и успех фреймворка Ruby on Rails связаны с тем, что силы Ruby-сообщества сосредоточены на единственной системе. Разрабатывается она давно, и за счет использования таких гемов, как `bundler`, `rake`, `Rack` и подобных, в нем удалось добиться удивительной совместимости и унификации. Усилия сообщества были сосредоточены на качественной проработке компонентов системы, а не на разработке одних и тех же компонентов под несовместимые друг с другом фреймворки.

Ruby on Rails — это не просто фреймворк, это объединение лучших гемов, которые сразу доступны «из коробки» (рис. 31.17).

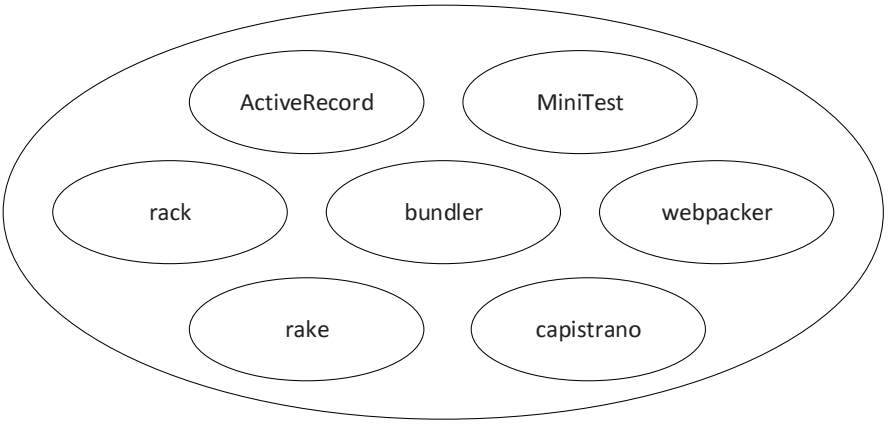


Рис. 31.17. Ruby on Rails — это набор гемов

Многие из гемов сами являются фреймворками для решения той или иной задачи:

- `bundler` — позволяет устанавливать гемы и управлять их зависимостями (см. *разд. 3.6.3*);
- `Rack` — позволяет взаимодействовать с серверами и другими Ruby-фреймворками (см. *разд. 31.3*);

- `rake` — позволяет связывать задачи в зависимые цепочки (см. *разд 3.4*);
- `capistrano` — позволяет автоматически доставлять и развертывать веб-приложения на серверы;
- `webpacker` — позволяет собирать и минифицировать ассеты (JS и CSS-файлы);
- `MiniTest` — позволяет тестировать приложение.

Гемы можно заменять друга на друга — например, если вам больше нравится `RSpec`, можно использовать его вместо `MiniTest` (рис. 31.18).

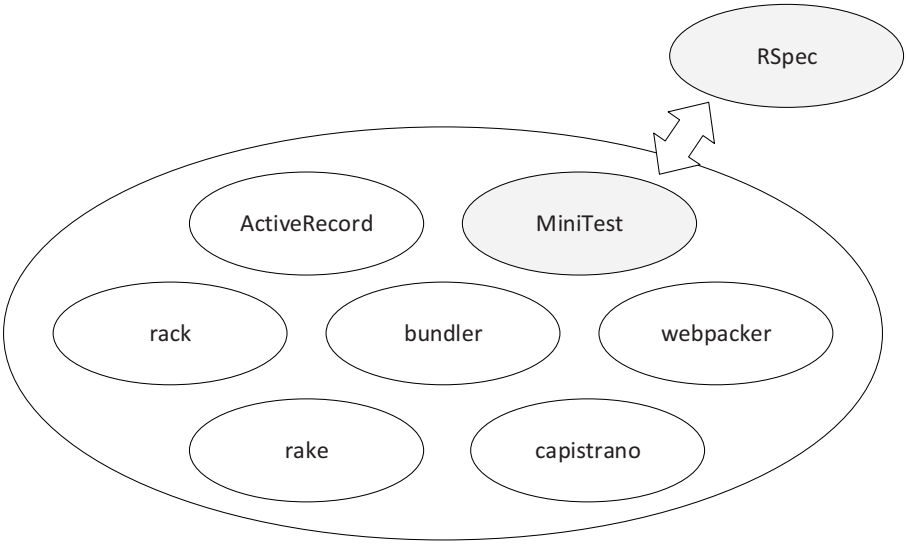


Рис. 31.18. Гемы можно заменять

Так можно поступать со всеми компонентами системы, т. е. Ruby on Rails — это своеобразный конструктор. Вы начинаете с набора подобранных и связанных друг с другом гемов и можете постепенно заменять компоненты на более вам подходящие.

31.4.2. Установка Ruby on Rails

Для того чтобы установить Ruby on Rails, лучше всего воспользоваться командой `gem install`:

```
$ gem install rails
Fetching thread_safe-0.3.6.gem
Fetching tzinfo-1.2.5.gem
...
Fetching sprockets-rails-3.2.1.gem
Successfully installed concurrent-ruby-1.1.4

HEADS UP! i18n 1.1 changed fallbacks to exclude default locale.
But that may break your application.
```

```
Please check your Rails app for 'config.i18n.fallbacks = true'.
If you're using I18n (>= 1.1.0) and Rails (< 5.2.2), this should be
'config.i18n.fallbacks = [I18n.default_locale]'.
If not, fallbacks will be broken in your app by I18n 1.1.x.

For more info see:
https://github.com/svenfuchs/i18n/releases/tag/v1.1.0

Successfully installed i18n-1.5.3
Successfully installed thread_safe-0.3.6
...
Successfully installed sprockets-rails-3.2.1
Successfully installed rails-5.2.2
Done installing documentation for concurrent-ruby, i18n, thread_safe,
tzinfo, activesupport, rack-test, mini_portile2, nokogiri, crass, loofah,
rails-html-sanitizer, rails-dom-testing, builder, erubi, actionview,
actionpack, activemodel, arel, activerecord, globalid, activejob, mini_mime,
mail, actionmailer, nio4r, websocket-extensions, websocket-driver, actioncable,
mimemagic, marcel, activestorage, thor, railties, sprockets, sprockets-rails,
rails after 41 seconds
36 gems installed
```

Мы не станем использовать `bundler`, т. к. он сам является зависимостью для гема `rails` и уже встроен в состав нашего будущего Rails-приложения.

Установив фреймворк, можно воспользоваться утилитой `rails` с тем, чтобы создать новое приложение. Для этого служит команда `rails new`. После команды указывается название приложения (в нашем случае `blog`) — приложение будет развернуто в одноименный каталог.

Во время установки создается первоначальная структура приложения и ставятся все зависимые гемы:

```
$ rails new blog
      create
      create  README.md
      create  Rakefile
      ...
      remove  config/initializers/cors.rb
      remove  config/initializers/new_framework_defaults_5_2.rb
         run  bundle install
The dependency tzinfo-data (>= 0) will be unused by any of the platforms
Bundler is installing for. Bundler is installing for ruby but the dependency is
only for x86-mingw32, x86-mswin32, x64-mingw32, java. To add those platforms to
the bundle, run `bundle lock --add-platform x86-mingw32 x86-mswin32 x64-mingw32
java`.
Fetching gem metadata from https://rubygems.org/.........
Fetching gem metadata from https://rubygems.org/.
Resolving dependencies...
Fetching rake 12.3.2
...
```

```
Fetching web-console 3.7.0
Installing web-console 3.7.0
Bundle complete! 18 Gemfile dependencies, 79 gems now installed.
Bundled gems are installed into `./vendor/bundle`
Post-install message from i18n:

HEADS UP! i18n 1.1 changed fallbacks to exclude default locale.
But that may break your application.

Please check your Rails app for 'config.i18n.fallbacks = true'.
If you're using I18n (>= 1.1.0) and Rails (< 5.2.2), this should be
'config.i18n.fallbacks = [I18n.default_locale]'.
If not, fallbacks will be broken in your app by I18n 1.1.x.

For more info see:
https://github.com/svenfuchs/i18n/releases/tag/v1.1.0
...
```

Приложение управляется утилитой `rails`, которую следует запускать в каталоге приложения. Поэтому перед тем, как ей воспользоваться, следует перейти в только что созданный каталог blog при помощи команды `cd`:

```
$ cd blog
```

Запустить приложение можно при помощи команды `rails server`, которая запустит сервер на порту `300`:

```
$ rails server
=> Booting Puma
=> Rails 5.2.2 application starting in development
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.12.0 (ruby 2.6.0-p0), codename: Llamas in Pajamas
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

Теперь, если обратиться по адресу **http://localhost:3000**, можно обнаружить стартовую страницу (рис. 3.19).

В консоли, где был запущен сервер, на каждое обращение выводится подробный журнальный вывод:

```
...
Started GET "/" for 127.0.0.1 at 2019-02-04 21:43:28 +0300
Processing by Rails::WelcomeController#index as HTML
  Rendering vendor/bundle/gems/railties-
5.2.2/lib/rails/templates/rails/welcome/index.html.erb
  Rendered vendor/bundle/gems/railties-
5.2.2/lib/rails/templates/rails/welcome/index.html.erb (6.4ms)
Completed 200 OK in 23ms (Views: 15.4ms | ActiveRecord: 0.0ms)
...
```

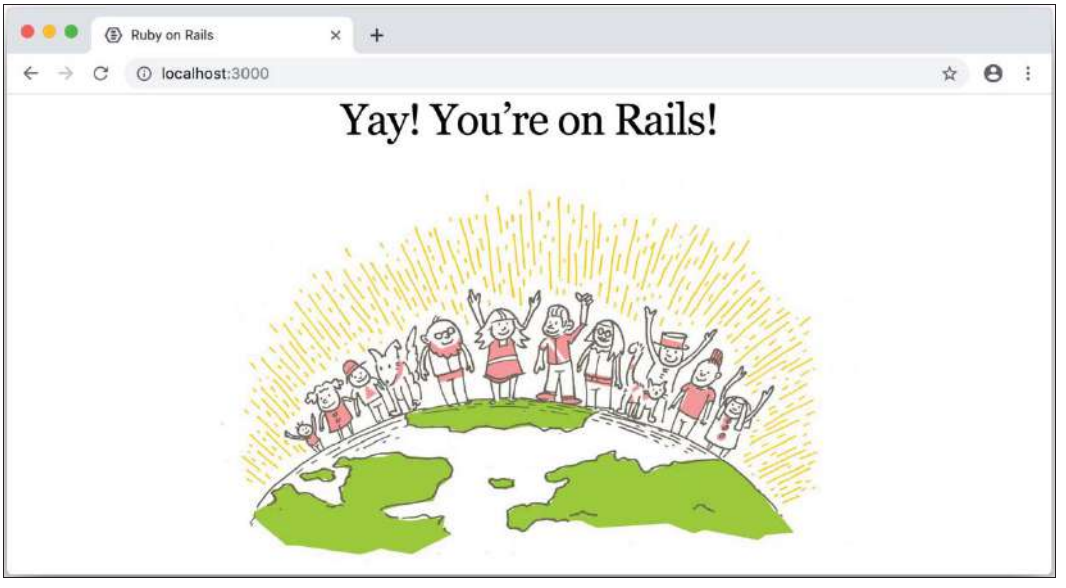


Рис. 31.19. Стартовая страница Ruby on Rails

Из журнального вывода можно извлечь практически всю информацию о запросе:

- `GET` — HTTP-метод;
- `"/"` — роут запроса;
- `2019-02-04 21:43:28 +0300` — время обращения к серверу;
- `Rails::WelcomeController#index` — класс контроллера и экшен (метод), которые обрабатывали запрос;
- `index.html.erb` — шаблоны, которые использовались для формирования страницы;
- `200` — HTTP-код ответа;
- `23ms` — время в миллисекундах, которое было затрачено на формирование страницы.

31.4.3. Паттерн MVC

Фреймворк Ruby on Rails поддерживает клиент-серверное взаимодействие (рис. 31.20).

Пользователь через браузер отправляет HTTP-запрос серверу. Сервер либо помещает полученную от пользователя информацию в базу данных, либо извлекает из базы данных сохраненную ранее информацию для формирования ответа. Затем HTTP-ответ отправляется браузеру, а он, в свою очередь, интерпретирует ответ сервера и отрисовывает веб-страницу.

Таким образом, веб-приложение всегда является *распределенным* (рис. 31.21). Часть приложения работает на сервере — она может обращаться к одной или не-

скольким базам данных. В браузере пользователя работает другая часть приложения, использующая клиентские технологии: язык разметки HTML, графическое оформление при помощи каскадных таблиц стилей CSS, язык программирования JavaScript. Клиентская часть может обращаться сразу к нескольким веб-серверам.

Все, что расположено на серверах, обычно называется *серверной*, или backend-частью приложения. Все, что работает в браузере, называется *клиентской*, или

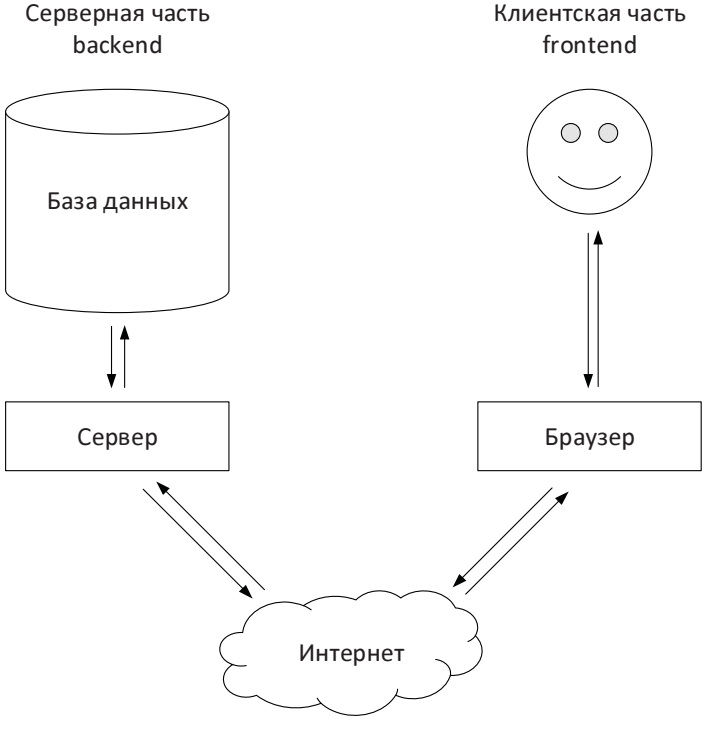


Рис. 31.20. Клиент-серверное взаимодействие

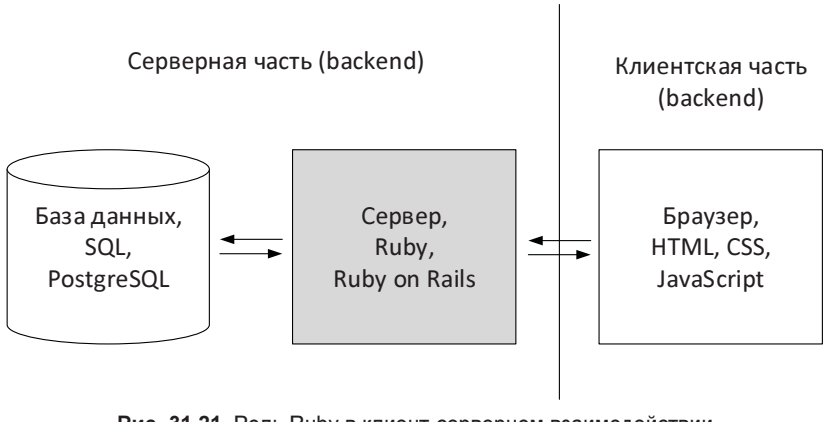


Рис. 31.21. Роль Ruby в клиент-серверном взаимодействии

frontend-частью приложения. Соответственно, разработчики также делятся на frontend- и backend-специалистов.

Ruby-код всегда выполняется только на сервере, т. к. браузер может интерпретировать лишь HTML-разметку, применять CSS-стили и выполнять код JavaScript. То есть при разработке сайта на Ruby on Rails мы, в основном, имеем дело с серверной частью, представленной на рис. 31.21 серым квадратом. Практически все, о чем мы будем говорить далее, будет в этом сером квадрате и сосредоточено.

Представленные на рис. 31.21 части сайта имеют собственную роль и специализацию:

- *база данных* осуществляет долговременное хранение данных: регистрацию пользователей, написанные ими тексты;
- *веб-сервер* обрабатывает поступающие запросы и формирует ответы, используя информацию из базы данных;
- *браузер* предоставляет ответ сервера в виде, доступном для просмотра человеком. Кроме того, в зону его ответственности входит формирование элементов управления для ввода информации.

Таким образом, информация, которой оперирует веб-приложение, находится в трех разных местах: в базе данных, на сервере и в браузере. Эти три части веб-приложения необходимо синхронизировать друг с другом. Если какая-либо информация (запись) появляется в базе данных, она должна стать доступной пользователю в браузере. Если пользователь что-то ввел в браузере и отправил эту информацию на сохранение, его сообщение должно попасть в базу данных. Если сообщение было скрыто администратором, а у пользователя нет прав на просмотр скрытых сообщений, оно не должно быть доступно для просмотра. Такой синхронизацией данных — координацией деятельности — занимается центральная часть приложения на Ruby on Rails. Для того чтобы упорядочить процесс синхронизации, прило-

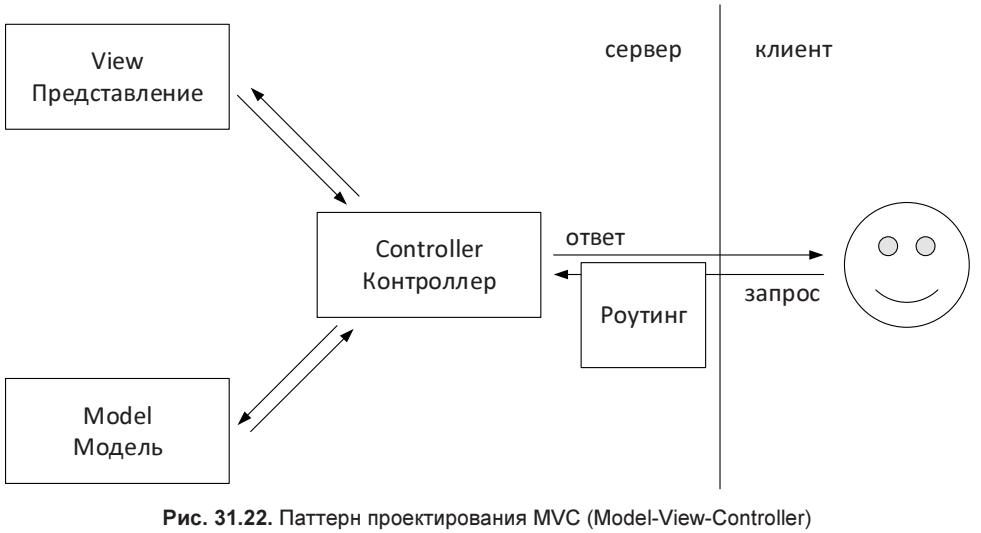


Рис. 31.22. Паттерн проектирования MVC (Model-View-Controller)

жение Ruby on Rails реализуется с использованием паттерна проектирования MVC: Модель (Model) — Представление (View) — Контроллер (Controller) (рис. 31.22).

Задача паттерна MVC заключается в специальной организации классов и объектов. Цель этой организации — разделить данные, представления и бизнес-логику. За данные отвечает модель, за HTML, CSS и JavaScript — представление, а за бизнес-логику — контроллер. Ruby on Rails дополнительно вводит подсистему роутинга, которая позволяет сопоставить запросы пользователя с тем или иным контроллером.

31.4.4. Структура приложения Ruby on Rails

Вслед за Ruby, в фреймворке Ruby on Rails большое внимание уделяется соглашениям. Фреймворк спроектирован таким образом, чтобы настройки по умолчанию подходили для подавляющего большинства случаев и практически не требовали вмешательства. В отличие от Ruby-приложений, в Ruby on Rails почти никогда не придется использовать метод `require` для подключения библиотек и файлов. Если следовать соглашениям расположения файлов и компонентов, Ruby on Rails всегда сможет обнаружить необходимые классы. Однако это требует понимания структуры приложения: за что отвечает тот или иной файл, где следует располагать новые файлы.

На рис. 31.23 представлена структура каталогов и файлов типичного приложения Ruby on Rails.

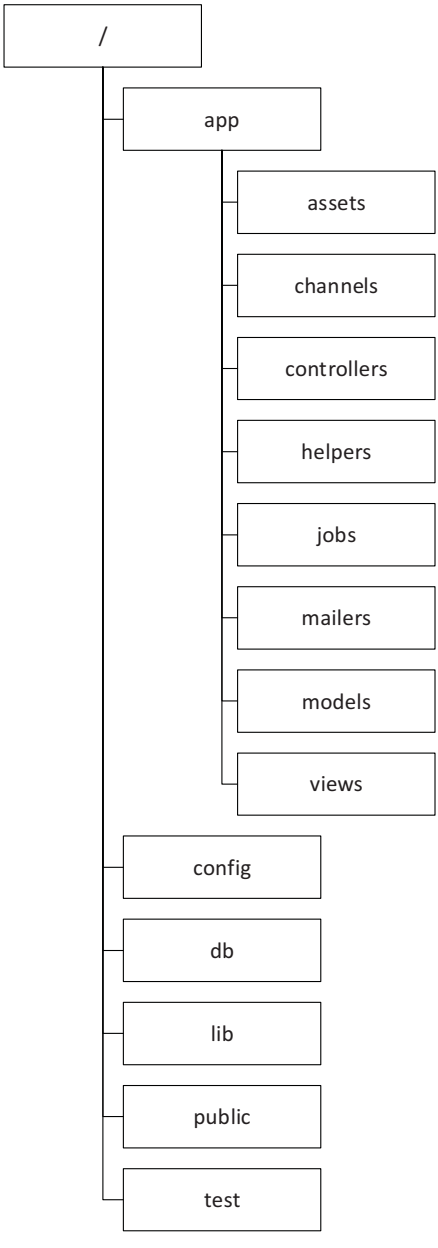


Рис. 31.23. Структура приложения Ruby on Rails

31.4.4.1. Каталог app

В каталоге app сосредотачивается весь код, созданный с использованием компонентов Ruby on Rails. Внутри каталога app располагаются несколько подкаталогов, среди которых следует выделить:

- models — содержит файлы с классами моделей;
- controllers — содержит файлы с классами контроллеров;
- views — содержит подкаталоги с шаблонами представлений.

Помимо этих основных подкаталогов, имеется еще несколько, содержащих вспомогательные компоненты:

- assets — ассеты, CSS, JavaScript-файлы, а также файлы изображений. Их использует CSS, также они необходимы в дизайне;
- channels — ActionCable-классы, необходимые для обслуживания WebSocket-соединений;
- helpers — небольшие вспомогательные методы, которые доступны в представлениях и позволяют убирать из них сложную или часто повторяющуюся логику;
- javascript — точка входа для JavaScript-приложения гема `Webpacker`;
- jobs — классы для создания фоновых задач в рамках гема `ActiveJob`, встроенного в Rails. Фактически, это интерфейс для единообразного оформления фоновых задач для целого класса гемов, реализующих очереди (`sidekiq`, `resque`, `que`, `delayed job`);
- mailers — обслуживание задач по отправке почтовых сообщений.

31.4.4.2. Окружения

Ruby on Rails при помощи гема `bundler` предоставляет несколько *окружений*. Сразу после установки нового приложения Ruby on Rails доступны три окружения:

- **development** — окружение по умолчанию, предназначенное для разработки. В этом окружении разработчик проводит бóльшую часть времени. Оно сконфигурировано таким образом, чтобы предоставить максимально удобную среду для разработки и отладки: более подробные логи, моментальная перезагрузка файлов без необходимости перезапуска сервера, отсутствие этапа сборки и компиляции ассетов, отсутствие минификации JS-кода и т. п.;
- **test** — тестовое окружение, предназначенное для выполнения автоматических тестов;
- **production** — продакшен-окружение, предназначенное для запуска приложения на конечном сервере. Это окружение сконфигурировано таким образом, чтобы добиться максимальной скорости работы приложения.

Помимо трех стандартных окружений, можно заводить собственные. Например, для сервера, предназначенного для демонстрации нового релиза заказчику, можно завести дополнительное окружение **preprod**. Если в технологической цепочке вы-

пуска нового релиза предусмотрено ручное тестирование, можно завести дополнительную тестовую площадку **staging**, на которой будут работать тестировщики.

31.4.4.3. Каталог config

Каталог config содержит конфигурационные файлы как самого приложения Ruby on Rails, так и всех сторонних гемов. Как правило, все конфигурационные файлы имеют разумные начальные настройки, требующие лишь минимального вмешательства.

Ряд конфигурационных файлов имеют несколько вариантов по количеству окружений, которые поддерживаются приложением. Например, в каталоге config/environments можно увидеть три файла: development.rb, test.rb и production.rb. Эти файлы содержат уникальные настройки для каждого из окружений.

Разбиение конфигурации по разным окружениям может быть реализовано на уровне структуры конфигурационных файлов. Например, внутри файла config/database.yml есть YAML-структура, которая задает параметры доступа к базе данных (логин, пользователь, пароль, имя базы данных) под каждое из окружений.

Важным файлом является routes.rb. В нем настраивается подсистема роутинга. Именно этот файл определяет, какой из контроллеров будет обслуживать тот или иной запрос от клиента.

31.4.4.4. Каталог db

Каталог db содержит все, что относится к базе данных:

- файлы миграций, позволяющие воссоздавать структуру базы данных на сервере, при установке приложения на новой машине разработчика или при тестировании;
- схему базы данных;
- seed-файл для первичного заполнения базы данных тестовыми или рабочими данными.

31.4.4.5. Каталог lib

Каталог lib предназначен для модулей и классов, которые не являются непосредственными компонентами Ruby on Rails или структуры MVC. Например, мы хотим сделать онлайн-игру «Морской бой». В игре есть множество классов и объектов Ruby, которые не имеют отношения к веб-программированию и Ruby on Rails: корабли, координаты, выстрел и т. д. Все эти вспомогательные классы, как правило, располагаются в каталоге lib, а не внутри каталога app, который обслуживает веб-приложение.

Здесь размещается код `Rake`-задач, которые разработчик создает для обслуживания приложения, базы данных, процедуры деплоя (развертывания веб-приложения на серверах) и т. п.

31.4.4.6. Каталог public

Каталог public является точкой входа приложения. Любой файл, который помещается в этот каталог, становится доступным на сайте. Внутри каталога можно найти несколько файлов-заготовок:

- 404.html — обработчик HTTP-кода 404: страница не найдена;
- 422.html — обработчик HTTP-кода 422: корректный запрос, который сервер не может обработать. Например index.json, если приложение не поддерживает отдачу в JSON-формате;
- 500.html — обработчик HTTP-кода 500: ошибка на сервере;
- robots.txt — файл с инструкциями для роботов поисковых систем.

Если запустить приложение при помощи команды `rails server`, то по адресу **http://localhost:3000/robots.txt** можно увидеть содержимое файла. Все файлы из каталога public доступны для просмотра через браузер.

31.4.4.7. Каталог test

Каталог test содержит все, что относится к тестам: сами тесты, вспомогательные хелперы и классы, обслуживающие тесты. Здесь же размещается код, упрощающий заполнение моделей базы данных перед выполнением тестов: фикстуры и фабрики.

31.4.5. *Rake*-задачи

Утилита `rails` поддерживает несколько типов задач, которые условно можно разделить на:

- `rails`-задачи генерации кода, запуска приложения и консоли;
- `rake`-задачи по обслуживанию приложения (см. *разд. 3.4*).

Чтобы познакомиться со списком задач, в корне проекта следует запустить команду `rails` без параметров:

```
$ rails
...
Rails:
  console
  ...
  server
  test
  version

Rake:
  ...
  assets:clobber
  assets:environment
  assets:precompile
  cache_digests:dependencies
```

```
cache_digests:nested_dependencies
db:create
db:drop
...
tmp:create
yarn:install
```

В предыдущих версиях Ruby on Rails задачи первой группы запускались утилитой `rails`, второй — `rake`. В настоящий момент все задачи можно выполнить при помощи `rails`. Список команд, который выдается командой `rails`, не является постоянным — команды могут быть добавлены и гемами, и самим разработчиком. Поэтому начинать знакомство с проектом следует со списка доступных команд.

У некоторых команд имеются сокращения — например, вместо `rails server` можно использовать `rails s` — сокращенную форму запуска сервера.

Помимо `Rails`-команд, доступны `Rake`-задачки, которые в отчете утилиты `rails` сосредоточены в секции `Rake`. В справке, которая выводит команда `rails`, `Rake`-задачки выводятся без описания. Однако, если мы воспользуемся командами `rake --task` или `rails -T`, можно получить список `Rake`-задач с подробным описанием:

```
$ rails -T
rails about                  # List versions of all Rails frameworks
rails active_storage:install # Copy over the migration needed to the app
...
rails tmp:create             # Creates tmp directories for cache, sockets
rails yarn:install           # Install all JavaScript dependencies
```

Можно использовать как готовые `Rake`-задачи, так и создавать свои собственные. Как и любой современный Ruby-фреймворк, Ruby on Rails — это `Rack`-приложение. При помощи команды `rake middleware` можно познакомиться со списком промежуточных слоев (см. *разд. 31.3.7*):

```
$ rails middleware
use Rack::Sendfile
use ActionDispatch::Static
use ActionDispatch::Executor
use ActiveSupport::Cache::Strategy::LocalCache::Middleware
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use ActionDispatch::RemoteIp
use Sprockets::Rails::QuietAssets
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use WebConsole::Middleware
use ActionDispatch::DebugExceptions
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
```

```
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ContentSecurityPolicy::Middleware
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
use Rack::TempfileReaper
run Blog::Application.routes
```

Как видим, любое приложение Ruby on Rails уже содержит большое количество готовых промежуточных слоев. Их несколько больше в окружении **development** и чуть меньше в **production**.

31.4.6. Генераторы

Ruby on Rails не только выполняет первоначальное развертывание заглушки, но и позволяет осуществить генерацию моделей, контроллеров и представлений. Для автогенерации нового кода предназначена специальная команда — `rails generate`. Чтобы узнать ее возможности, команду следует запустить без дополнительных параметров. Вместо полного варианта `rails generate` можно использовать сокращенный — `rails g`:

```
$ rails g
Rails:
  application_record
  ...
  model
  resource
  scaffold
  scaffold_controller
  system_test
  task
  ...
```

Как видим, представлено большое количество самых разнообразных генераторов. Давайте воспользуемся генератором `rails g task`, который позволяет создать заглушку для новой `Rake`-задачки. Для того чтобы получить справочную информацию по этой команде, ее достаточно запустить без дополнительных параметров:

```
$ rails g task
Running via Spring preloader in process 39204
Usage:
  rails generate task NAME [action action] [options]

Options:
  [--skip-namespace], [--no-skip-namespace]  # Skip namespace (affects only
                                                          isolated applications)
```

```
Runtime options:
  -f, [--force]                    # Overwrite files that already exist
  -p, [--pretend], [--no-pretend]  # Run but do not make any changes
  -q, [--quiet], [--no-quiet]      # Suppress status output
  -s, [--skip], [--no-skip]        # Skip files that already exist

Description:
    Stubs out a new Rake task. Pass the namespace name, and a list of tasks
                                                                  as arguments.

    This generates a task file in lib/tasks.

Example:
    `rails generate task feeds fetch erase add`

        Task:      lib/tasks/feeds.rake
```

В ответ выводится подробное описание команды, а в секции `Example` можно найти пример ее использования. Создадим простейшую `Rake`-задачу в пространстве имен `example`:

```
$ rails g task example
Running via Spring preloader in process 39399
      create  lib/tasks/example.rake
```

В отчете команды `rails generate` всегда выводится список созданных файлов и их расположение. `Rake`-задачи находятся по пути lib/task — здесь должен находиться только что созданный файл example.rake (листинг 31.11).

Листинг 31.11. Файл blog/lib/tasks/example.rake (предварительно)

```
namespace :example do
end
```

В файле пока прописан только вызов метода `namespace`. В листинге 31.12 представлен модифицированный файл example.rake с задачей `hello`.

Листинг 31.12. Файл blog/lib/tasks/example.rake (окончательно)

```
namespace :example do
  desc 'Пример rake-задачи'
  task :hello do
    puts 'Hello, world!'
  end
end
```

Теперь, если перейти в консоль и запустить команду `rails -T`, среди списка `Rake`-задач можно обнаружить новую задачу `example:hello`:

```
$ rails -T
rails about                  # List versions of all Rails frameworks
rails active_storage:install # Copy over the migration needed to the app
...
rails example:hello          # Пример rake-задачи
...
rails tmp:create             # Creates tmp directories for cache, sockets
rails yarn:install           # Install all JavaScript dependencies
```

Задачу `example:hello` можно запустить на выполнение:

```
$ rake example:hello
Hello, world!
```

Rake-задачу можно было создать и без генератора. Все, что делают генераторы, можно выполнять вручную, однако генераторы значительно ускоряют процесс разработки.

Поведение генераторов можно настроить. Для этого в конфигурационном файле config/environments/development.rb можно вызывать метод `generators`. Его блок позволяет настраивать, какие файлы будут создаваться при вызове генераторов (листинг 31.13).

Листинг 31.13. Файл blog/config/environments/development.rb

```
Rails.application.configure do
  ...
  config.generators do |g|
    g.orm             :active_record
    g.test_framework  false
    g.helper          false
    g.stylesheets     false
    g.javascripts     false
  end
end
```

Инструкции в приведенном примере запрещают генерацию тестов, хелперов, каскадных таблиц стилей и JavaScript-файлов.

31.4.7. Стартовая страница

В этом разделе мы заменим главную страницу сайта (см. рис. 31.19) своей собственной и воспользуемся для этого генератором контроллера. Контроллер необходим для обработки запросов клиентов к веб-приложению — это точка входа в схеме MVC (см. рис. 31.20).

Rails-контроллер — это класс, в котором определены методы-экшены (действия). Последним сопоставляются роуты, которые вводятся в адресной строке браузера.

С помощью команды `rails generate controller` можно создать контроллер и представления. Набрав команду без аргументов, можно ознакомиться со справкой по ее использованию:

```
$ rails g controller
Usage:
  rails generate controller NAME [action action] [options]
...
Example:
    `rails generate controller CreditCards open debit credit close`

    CreditCards controller with URLs like /credit_cards/debit.
        Controller: app/controllers/credit_cards_controller.rb
        Test:       test/controllers/credit_cards_controller_test.rb
        Views:      app/views/credit_cards/debit.html.erb [...]
        Helper:     app/helpers/credit_cards_helper.rb
```

В секции `Example` приводится пример создания контроллера `CreditCards` с четырьмя экшенами: `open`, `debit`, `credit` и `close`. Для стартовой страницы можно создать контроллер `Home` с единственным экшеном `index`:

```
$ rails g controller Home index
      create  app/controllers/home_controller.rb
       route  get 'home/index'
      invoke  erb
      create    app/views/home
      create    app/views/home/index.html.erb
      ...
```

Как можно видеть, генератор формирует контроллер в каталоге app/controllers, запись в файле config/routes.rb с роутингом для контроллера, а также шаблоны представлений в каталоге app/views.

Когда запрос поступает приложению Ruby on Rails, после его первичной обработки он передается в подсистему роутинга. Управление роутингом сосредоточено в файле config/routes.rb (листинг 31.14).

Листинг 31.14. Файл blog/config/routes.rb (предварительно)

```
Rails.application.routes.draw do
  get 'home/index'
end
```

Сейчас в файле записан только один роут — в виде метода `get`, которому передается путь `'home/index'`.

Метод `get` сопоставляет `GET`-запрос по адресу **http://localhost:3000/home/index** с методом `index` контроллера `Home`.

Контроллеры, как уже отмечалось, сосредоточены в каталоге apps/controllers — здесь можно обнаружить созданный генератором файл home_controller.rb (листинг 31.15).

Листинг 31.15. Файл blog/app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
  end
end
```

Внутри файла прописан класс `HomeController`, который был создан генератором. Как можно видеть, наследуется он от класса `ApplicationController`, который расположен тут же — в каталоге apps/controllers (листинг 31.16).

Листинг 31.16. Файл blog/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
end
```

`ApplicationController` — это базовый класс для всех контроллеров веб-приложения. Он наследуется от класса `ActionController::Base`, являющегося компонентом Ruby on Rails.

Возвращаясь к `HomeController` (см. листинг 31.15), можно заметить, что в нем определен метод `index`. Этот метод является экшеном — одной из точек входа веб-приложения, которой может быть сопоставлен один или несколько роутов.

Если сейчас перейти по адресу **http://localhost:3000/home/index**, то в консоли сервера можно будет обнаружить журнальную запись следующего вида:

```
Started GET "/home/index" for 127.0.0.1 at 2019-02-06 09:33:49 +0300
Processing by HomeController#index as HTML
  Rendering home/index.html.erb within layouts/application
  Rendered home/index.html.erb within layouts/application (1.2ms)
Completed 200 OK in 381ms (Views: 374.0ms | ActiveRecord: 0.0ms)
```

В первой строке можно видеть обращение к роуту /home/index, во второй строке сообщается, что за обработку запроса отвечает контроллер `HomeController` и его экшен `index`.

Один и тот же экшен может обслуживать запросы от множества роутов. Для этого в файл config/routes.rb необходимо добавить новые роуты, которые будут указывать на экшен `index` контроллера `HomeController`. Это можно сделать добавлением вызова метода `get` и передачей ему дополнительного параметра `to:` со значением `'home#index'`. Так, в листинге 31.17 формируется роут для адреса **http://localhost:3000/hello**.

Листинг 31.17. Дополнительный роут. Файл blog/config/routes.rb (предварительно)

```
Rails.application.routes.draw do
  get 'home/index'
  get 'hello', to: 'home#index'
end
```

Если необходимо, чтобы контроллер обслуживал главную страницу, можно воспользоваться специальным методом `root` (листинг 31.18).

Листинг 31.18. Использование метода `root`. Файл blog/config/routes.rb

```
Rails.application.routes.draw do
  get 'home/index'
  get 'hello', to: 'home#index'
  root 'home#index'
end
```

Теперь на главной странице вместо приветственной заглушки фреймворка Ruby on Rails будет отображаться результат работы контроллера `HomeController`.

31.4.8. Представления

Помимо контроллера, для формирования страницы необходимы шаблоны представления. Они сосредоточены в каталоге views (рис. 31.24).

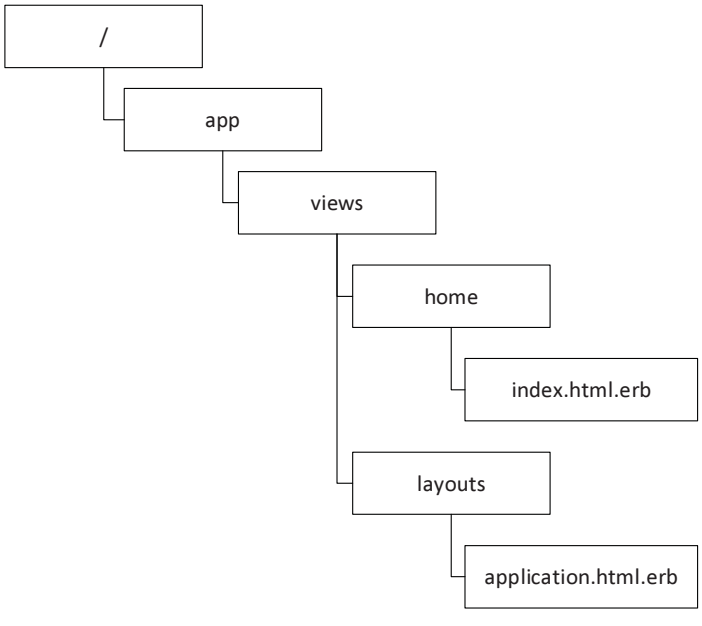


Рис. 31.24. Схема представлений

По пути app/views/home/index.html.erb можно обнаружить заготовку шаблона представления для экшена `HomeController#index` (листинг 31.19).

Листинг 31.19. Файл blog/app/views/home/index.html.erb (предварительно)

```
<h1>Home#index</h1>
<p>Find me in app/views/home/index.html.erb</p>
```

Обработкой такого шаблона занимается стандартная библиотека `Erb` языка Ruby (см. *разд. 3.3*).

ERB-шаблоны позволяют при помощи специальных тэгов `<%= ... %>` вставлять любой Ruby-код. В листинге 31.20 вычисляется выражение `2 + 2`, которое помещается в HTML-параграф `<p>...</p>`. Последнее необходимо, чтобы результирующее выражение на HTML-странице было размещено на отдельной строке.

> ***Замечание***
>
> Язык разметки HTML используется для формирования структуры веб-страницы. Этот декларативный язык хотя и прост, все же требует отдельного изучения. Его детальное рассмотрение выходит за рамки этой книги.

Листинг 31.20. Файл blog/app/views/home/index.html.erb (окончательно)

```
<h1>Home#index</h1>
<p>Find me in app/views/home/index.html.erb</p>
<p><%= 2 + 2 %></p>
```

Теперь страница, которую обслуживает это представление, будет выглядеть так, как представлено на рис. 31.25.

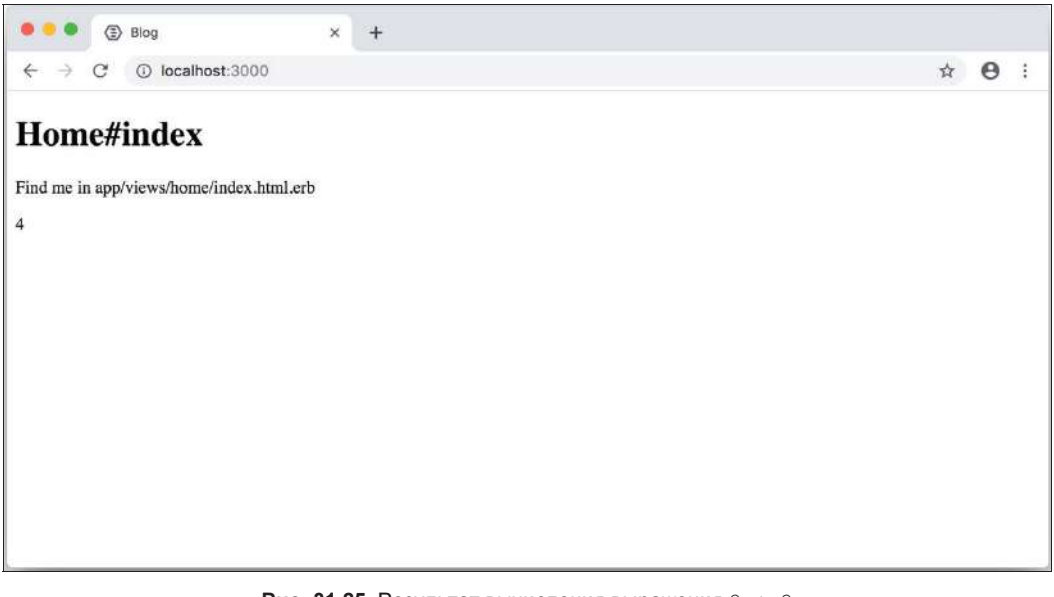


Рис. 31.25. Результат вычисления выражения `2 + 2`

Так как контроллер называется `HomeController`, его представления сосредоточены в каталоге app/views/home. Экшен называется `index`, поэтому и файл с шаблоном называется index.html.erb. Если файлы и каталоги будут называться по-другому, все сломается. Для исправления ситуации потребуется явно прописать вызов метода `render` в экшене `HomeController#index` (листинг 31.21).

Листинг 31.21. Явный вызов метода `render` в экшене

```
class HomeController < ApplicationController
  def index
    render 'home/index'
  end
end
```

Однако, если файлы и каталоги названы в соответствии с соглашениями, принятыми в Ruby on Rails, можно не указывать явный вызов метода `render`. И поскольку его можно не указывать, он обычно никогда и не указывается.

Ruby on Rails построен на многочисленных соглашениях. С одной стороны, для работы с фреймворком нужно изучать и знать эти соглашения, с другой стороны, благодаря им код становится очень компактным.

Когда вы только начинаете изучать Ruby on Rails, очень легко совершить ошибки, нарушив какое-либо из соглашений. Поэтому на начальных этапах важно использовать генераторы, которые правильно назовут файлы и не позволят ошибиться.

В короткой главе невозможно охватить все возможности Ruby on Rails. Однако использовать этот фреймворк гораздо проще и эффективнее по трудозатратам, чем писать свое собственное `Rack`-приложение.

Задания

1. Реализуйте `Rack`-приложение, при обращении к страницам которого в HTTP-ответе станет отправляться HTTP-заголовок с именем `X-Ruby-Version`, в значении которого будет передаваться текущая версия Ruby.
2. В приложении Ruby on Rails создайте `Rake`-задачу, которая бы подсчитывала количество Ruby-файлов в проекте.
3. В приложении Ruby on Rails создайте `Rake`-задачу, которая бы подсчитывала количество строк в Ruby-файлах проекта.
4. Реализуйте на Ruby on Rails сайт с главной страницей `/` и страницей «О нас» `/about`. На каждой странице должно быть меню, которое позволит переходить с одной страницы на другую.
5. При помощи механизма сессий и хэша `session` организуйте в приложении Ruby on Rails подсчет количества посещений страницы пользователем. Перезагрузка страницы должна приводить к увеличению счетчика посещений.
6. Инсталлируйте базу данных PostgreSQL и установите с ней соединение из приложения Ruby on Rails. Средствами Ruby on Rails создайте таблицу пользователей `users`, которая будет хранить имена и фамилии пользователей. Создайте модель `User` для работы с таблицей и заполните ее из Rails-консоли. Разработайте `Rake`-задачу, которая выводит список пользователей из таблицы `users` в алфавитном порядке.
7. Изучите веб-фреймворк Sinatra, выведите с его помощью на главной странице сайта фразу `'Hello, world!'`.

ГЛАВА 32

Автоматическое тестирование

Файлы с исходными кодами этой главы находятся в каталоге *tests* сопровождающего книгу электронного архива.

Автоматическое тестирование играет в Ruby-сообществе заметную роль. Ruby-инструменты и фреймворки тестирования настолько разнообразны и отточены, что используются широко за пределами Ruby-сообщества.

В этой главе мы познакомимся с автоматическим тестированием и рассмотрим три популярных фреймворка для тестирования: `MiniTest`, `RSpec` и `Cucumber`.

Тестирование уже давно стало в IT-индустрии независимым направлением. Поэтому как самому тестированию, так и каждому из упомянутых в этой главе фреймворков можно посвятить отдельную книгу.

32.1. Типы тестирования

Тесты условно поделить на несколько групп:

- *unit-тесты* предназначены для тестирования участка кода: какого-либо метода или утверждения;
- *интеграционные тесты* проверяют какую-либо функциональность или реакцию: зашли на страницу, нажали кнопку, получили результат;
- *приемочные тесты* предназначены, скорее, для аналитиков, прожект-менеджеров и руководителей групп. Они пишутся заранее и задают какое-либо свойство или поведение, которое обязательно должно присутствовать в конечной программе;
- *ручное тестирование* применяется в том случае, когда задачи очень сложно тестировать автоматически, — например, верстку сайта, расположение кнопок, следование требованиям. Эту работу зачастую выполняют вручную. Кроме того, автоматические тесты редко покрывают все возможные случаи. Поэтому ручное тестирование должно вскрывать новые и неизвестные ошибки. Как правило, по выявленным проблемам составляются *тест-кейсы* — текстовые описания проблем и методов их воспроизведения. Наиболее часто встречающиеся случаи подлежат автоматизации в виде интеграционных тестов.

Рис. 32.1. Типы тестирования

Обратите внимание на пирамиду, показанную на рис. 32.1, — она отражает примерное количественное соотношение между разными типами тестов.

- Unit-тестов должно быть максимально много — они быстро выполняются и позволяют вскрывать низкоуровневые ошибки.
- Интеграционных тестов может быть меньше — они выполняются уже гораздо дольше по времени и должны покрывать только важный функционал.
- Приемочных тестов должно быть еще меньше — они не должны диктовать реализацию программы, их задача — задать конечный результат: поведение, без которого программа не может считаться законченной. Кроме того, очень трудно что-то запрограммировать, не имея на руках прототип или готовую программу. Приемочные тесты — это, зачастую, тесты, созданные до того, как началась работа над самой программой.
- Ручного тестирования должно быть еще меньше — даже очень трудолюбивый человек не может тягаться по скорости и методичности с компьютерной программой. Если тестировщик будет проверять программу после каждого изменения в коде, он рано или поздно либо не справится, либо станет ошибаться. В идеальном случае задача тестировщика руководить всем процессом тестирования: писать тест-кейсы, находить часто возникающие ошибочные ситуации, тестировать те случаи, которые трудно поддаются автоматизации.

32.2. Преимущества и недостатки тестирования

Пока тесты не используются, их преимущества не кажутся очевидными. Тем не менее они дают контроль над правильностью выполнения программы. Если проект покрыт тестами, то при внесении ошибочных изменений в код тесты будут сигнализировать, что изменения ломают какую-то часть ранее созданного функционала.

Создание тестов может рассматриваться как дополнительный анализ кода с целью улучшения его архитектуры, стиля, поиска ошибок. Причем его невозможно про-

вести формально — при помощи тестов приходится доказывать, что код работает правильно. Кроме того, тесты отлично вскрывают плохо написанную или спроектированную программу. Если тесты писать сложно — с кодом явно что-то не так, это сигнал для его переработки.

Разумеется, у тестов есть и обратная сторона медали. Тесты помогают снизить количество ошибок, но не гарантируют их отсутствие. Высокое покрытие тестами может создать иллюзию, что программа работает правильно. Однако полное доказательство правильной работы программы потребует усилий и кода на порядки больше, чем требуется для создания самой программы.

Тесты — это тоже код, они требуют времени на создание и сопровождение. И их не так легко создавать — потребуются усилия и время на то, чтобы этому научиться.

32.3. Фреймворки для тестирования

Ruby-сообщество фанатически относится к тестированию. Более того, Ruby-фреймворки часто используются для тестирования программ на других языках. Наиболее популярными фреймворками являются:

- `MiniTest` — тесты на нем организованы в виде классов и методов, которые начинаются с префикса `Test`. Этот фреймворк входит по умолчанию в Ruby on Rails, поэтому весьма популярен;
- `RSpec` — альтернативный фреймворк, который реализует декларативный DSL-язык тестирования, предметно-ориентированный язык, полностью посвященный тестированию;
- `Cucumber` — фреймворк, который создан для создания приемочных тестов. Его тесты представляют вместе с ними спецификацию на естественном языке, например на русском. Предполагается, что тесты и спецификация создаются совместно разработчиками, аналитиками и менеджерами проектов.

На практике чаще встречаются `MiniTest` и `RSpec` — это «рабочие лошадки» Ruby-сообщества.

32.3.1. Фреймворк *MiniTest*

Чтобы воспользоваться фреймворком `MiniTest`, установим его через гем `bundler` (см. *разд. 3.6.3*). Для создания заготовки конфигурационного файла Gemfile можно воспользоваться командой `bundle init`:

```
$ bundle init
Writing new Gemfile to /home/i.simdyanov/minitest/Gemfile
```

Впрочем, Gemfile можно создать любым другим способом. Внутри файла необходимо подключить гем `MiniTest` (листинг 32.1).

Листинг 32.1. Конфигурационный файл Gemfile. Файл minitest/Gemfile

```
source 'https://rubygems.org'
gem 'minitest'
```

Чтобы его установить, необходимо выполнить команду `bundle` без дополнительных параметров:

```
$ bundle
Fetching gem metadata from https://rubygems.org/.............
Resolving dependencies...
Using bundler 1.16.0
Fetching minitest 5.11.3
Installing minitest 5.11.3
Bundle complete! 1 Gemfile dependency, 2 gems now installed.
Bundled gems are installed into `./vendor/bundle`
```

Для тестирования используются *утверждения* — специальные методы, которые предоставляет фреймворк `MiniTest` и которые, как уже отмечалось, начинаются с префикса `test`. Как и любые методы в языке Ruby, их названия оформляются в snake-режиме. Название метода должно отражать тестируемое свойство или поведение.

Протестируем класс `Hello` с единственным методом `say`. Метод принимает параметр и возвращает фразу вида `'Hello, world!'`, если в параметр передано значение `'world'`. Пока класс `Hello` намеренно оставим незавершенным (листинг 32.2).

Листинг 32.2. Класс для тестирования `Hello`. Файл minitest/hello.rb (предварительно)

```
class Hello
  def say(str)
  end
end
```

Подключение `MiniTest` осуществляется с помощью метода `require`, которому передается строка `'minitest/autorun'`. Для тестирования создается класс, который начинается с префикса `Test` и наследуется от класса `MiniTest::Unit::TestCase`. В созданном классе переопределяется метод `setup`, который выполняется до начала всех тестов. В этом методе определена инстанс-переменная `@object`, которая инициализируется объектом класса `Hello` (листинг 32.3). В примере проверяется, что метод `say` возвращает строковое значение.

Листинг 32.3. Класс для тестирования `Hello`. Файл minitest/hello.rb (предварительно)

```
require 'minitest/autorun'

class Hello
  def say(str)
  end
end
```

```
class TestHello < MiniTest::Unit::TestCase
  def setup
    @object = Hello.new
  end

  def test_that_hello_return_a_string
    assert_instance_of String, @obj.say('test')
  end
end
```

При помощи утверждения `assert_instance_of` здесь проверяется, является ли возвращаемое методом `say` значение строкой, т. е. объектом класса `String`. Сейчас метод `say` возвращает значение `nil`, поэтому тест должен «упасть». Убедиться в этом можно, запустив программу на выполнение:

```
$ ruby hello.rb
MiniTest::Unit::TestCase is now Minitest::Test. From hello.rb:8:in `<main>'
Run options: --seed 57925

# Running:

F

Failure:
TestHello#test_that_hello_return_a_string [hello.rb:14]:
Expected nil to be an instance of String, not NilClass.

bin/rails test hello.rb:13

Finished in 0.001234s, 810.5882 runs/s, 810.5882 assertions/s.

1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

Как можно видеть, ожидается объект класса `String`, а вместо этого получено значение `nil`. Для того чтобы тест успешно выполнялся, необходимо исправить метод `say` класса `Hello` (листинг 32.4).

Листинг 32.4. Исправление класса `Hello`. Файл minitest/hello.rb (предварительно)

```
require 'minitest/autorun'

class Hello
  def say(str)
    "Hello, #{str}!"
  end
end
...
```

Теперь повторный запуск программы приводит к тому, что тест успешно проходит:

```
$ ruby hello.rb
MiniTest::Unit::TestCase is now Minitest::Test. From hello.rb:9:in `<main>'
Run options: --seed 17389

# Running:

.

Finished in 0.001197s, 835.5182 runs/s, 835.5182 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Как можно видеть, вместо сообщения об ошибке выводится одна точка — наш единственный тест прошел успешно.

> ***Замечание***
>
> Документацию к фреймворку `MiniTest` можно найти по ссылке **https://ruby-doc.org/stdlib-2.1.4/libdoc/minitest/rdoc/MiniTest/Assertions.html**.

В документации к `MiniTest` можно обнаружить большое количество самых разнообразных утверждений — методов, которые начинаются с префикса `assert_`. Самым простым из них является `assert_equal`, сравнивающий равенство одного утверждения с другим. В листинге 32.5 приводится еще один тест `test_that_hello_return_correct_phrase` с использованием этого утверждения.

Листинг 32.5. Исправление класса `Hello`. Файл minitest/hello.rb (окончательно)

```
require 'minitest/autorun'

class Hello
  def say(str)
    "Hello, #{str}!"
  end
end

class TestHello < MiniTest::Unit::TestCase
  def setup
    @object = Hello.new
  end

  def test_that_hello_return_a_string
    assert_instance_of String, @object.say('test')
  end

  def test_that_hello_return_correct_phrase
    assert_equal 'Hello, world!', @object.say('world')
  end
end
```

В этом тесте проверяется, вернет ли вызов метода `say` с передачей ему в качестве аргумента строки `'world'` строку `'Hello, world!'`. Теперь, если запустить программу на выполнение, можно получить следующие результаты:

```
$ ruby hello.rb
MiniTest::Unit::TestCase is now Minitest::Test. From hello.rb:9:in `<main>'
Run options: --seed 51569

# Running:

..

Finished in 0.001046s, 1911.5141 runs/s, 1911.5141 assertions/s.

2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Как можно видеть, здесь выводятся две точки — два пройденных теста.

32.3.2. Фреймворк *RSpec*

Для того чтобы воспользоваться фреймворком `RSpec`, необходимо установить гем `rspec`. В листинге 32.6 приводится файл Gemfile с подключенным гемом.

Листинг 32.6. Установка RSpec. Файл rspec/Gemfile

```
source 'https://rubygems.org'
gem 'rspec'
```

Для установки гема необходимо запустить команду `bundle` без параметров:

```
$ bundle
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using bundler 1.16.0
Fetching diff-lcs 1.3
Installing diff-lcs 1.3
Fetching rspec-support 3.8.0
Installing rspec-support 3.8.0
Fetching rspec-core 3.8.0
Installing rspec-core 3.8.0
Fetching rspec-expectations 3.8.2
Installing rspec-expectations 3.8.2
Fetching rspec-mocks 3.8.0
Installing rspec-mocks 3.8.0
Fetching rspec 3.8.0
Installing rspec 3.8.0
Bundle complete! 1 Gemfile dependency, 7 gems now installed.
Bundled gems are installed into `./vendor/bundle`
```

Для тестов создадим отдельный каталог spec, в котором будут располагаться файлы с тестами. Такие файлы должны иметь суффикс _spec.rb — именно на них будет реагировать утилита rspec, которая запускает тесты.

В листинге 32.7 представлен файл example_spec.rb с первым тестом. RSpec-тесты выглядят как декларативный язык — человек, не знакомый с Ruby, может читать их как обычные выражения.

Листинг 32.7. RSpec-тест. Файл rspec/spec/example_spec.rb (предварительно)

```
RSpec.describe 'Пример' do
  it 'должен работать когда' do
    expect(false).to be_falsy
  end
end
```

Тесты оборачиваются в метод describe класса RSpec. Метод принимает блок, внутри которого можно размещать тесты. Тесты задаются методом it, который тоже принимает блок. Внутри этого блока можно размещать expect-утверждения.

В приведенном примере утверждение не затейливое: тест считается пройденным, если объект false с логической точки зрения будет ложным. Пример несколько надуманный, но демонстрирует, как выглядят RSpec-тесты.

Для запуска тестов используется утилита rspec, которая была установлена вместе с гемом. Утилите в качестве аргумента передается путь до каталога с тестами — в нашем случае это каталог spec:

```
$ rspec spec
.

Finished in 0.00538 seconds (files took 0.1307 seconds to load)
1 example, 0 failures
```

Как можно видеть, RSpec сообщает, что единственный обнаруженный тест успешно выполнен.

В методе describe описывается, что тестируется, в методе it реализуются тесты. Каждый вызов метода it является отдельным тестом. Добавим в файл example_spec.rb еще один тест (листинг 32.8).

Листинг 32.8. Файл rspec/spec/example_spec.rb (предварительно)

```
RSpec.describe 'Пример' do
  it 'должен работать когда' do
    expect(false).to be_falsy
  end
  it 'должен работать когда' do
    expect(true).to be_truthy
  end
end
```

Теперь в отчете команды rspec можно обнаружить два пройденных теста:

```
$ rspec spec
..

Finished in 0.00545 seconds (files took 0.12742 seconds to load)
2 examples, 0 failures
```

Команда rspec может принимать параметры — например, при помощи параметра:

```
--format d
```

можно получить описание, которое было задано в методах describe и it:

```
$ rspec spec --format d

Пример
  должен работать когда
  должен работать когда

Finished in 0.00273 seconds (files took 0.09838 seconds to load)
2 examples, 0 failures
```

В идеале описания должны складываться в законченные предложения (листинг 32.9).

Листинг 32.9. Файл rspec/spec/example_spec.rb (предварительно)

```
RSpec.describe 'Пример' do
  it 'должен работать в случае false' do
    expect(false).to be_falsy
  end
  it 'должен работать в случае true' do
    expect(true).to be_truthy
  end
end
```

При запуске обновленного варианта тестов в выводе можно обнаружить полноценное описание тестируемых особенностей на русском языке:

```
$ rspec spec --format d

Пример
  должен работать в случае false
  должен работать в случае true

Finished in 0.00303 seconds (files took 0.13336 seconds to load)
2 examples, 0 failures
```

RSpec-тесты, помимо проверки правильности утверждений, позволяют создать спецификацию программы.

До текущего момента мы, в основном, имели дело только с успешно выполняющимися тестами. Давайте посмотрим, как тесты «падают». Для этого во втором тесте в листинге 32.9 исправим `true` на `false`:

```
...
  it 'должен работать в случае true' do
    expect(false).to be_truthy
  end
...
```

В этом случае команда `rspec` выдаст следующие результаты:

```
$ rspec spec
.F

Failures:

  1) Пример должен работать в случае true
     Failure/Error: expect(false).to be_truthy

       expected: truthy value
            got: false
     # ./spec/example_spec.rb:6:in `block (2 levels) in <top (required)>'

Finished in 0.02092 seconds (files took 0.09666 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/example_spec.rb:5 # Пример должен работать в случае true
```

При «падении» теста одна из точек меняется на букву `F` (от `false`), а ниже выводится подробное описание: какой тест «упал», что ожидалось и что было возвращено реально.

> ***Замечание***
>
> Документацию к фреймворку `RSpec` можно найти по ссылке **https://github.com/rspec/rspec-core**. Утверждения приводятся на странице **https://github.com/rspec/rspec-expectations**.

Помимо утверждения `expect`, фреймворк `RSpec` предоставляет большое количество других самых разнообразных утверждений.

В листинге 32.10 используется утверждение `be_beetween`, которое проверяет вхождение числа в диапазон.

Листинг 32.10. Утверждение `be_between`. Файл rspec/spec/between_spec.rb

```
RSpec.describe 'Число' do
  it 'должно входить в диапазон' do
```

```
    expect(5).to be_between(1, 10)
  end
end
```

В приведенном примере тест проверяет вхождение числа `5` в диапазон от `1` до `10`. Если запустить команду `rspec` на выполнение, можно убедиться, что все созданные к текущему моменту тесты проходят корректно:

```
$ rspec spec
...

Finished in 0.00687 seconds (files took 0.12168 seconds to load)
3 examples, 0 failures
```

Можно запускать не все тесты разом, а конкретные файлы — для этого вместо каталога spec следует указать путь к файлу:

```
$ rspec spec/example_spec.rb
..

Finished in 0.00408 seconds (files took 0.10513 seconds to load)
2 examples, 0 failures
```

В результате запускаются только те тесты, которые расположены внутри файла example_spec.rb. Более того, можно запустить один конкретный тест, если через двоеточие указать номер строки, на которой он расположен:

```
$ rspec spec/example_spec.rb:5
Run options: include {:locations=>{"./spec/example_spec.rb"=>[5]}}
.

Finished in 0.00252 seconds (files took 0.14049 seconds to load)
1 example, 0 failures
```

Если тест пока не реализован, его можно объявить, как pending-тест (отложенный тест). Для этого вместо метода `it` используется метод `xit` (листинг 32.11).

Листинг 32.11. Отложенный тест. Файл rspec/spec/example_spec.rb (окончательно)

```
RSpec.describe 'Пример' do
  it 'должен работать в случае false' do
    expect(false).to be_falsy
  end
  it 'должен работать в случае true' do
    expect(true).to be_truthy
  end
  xit 'отложен' do
    # TODO: Реализовать тест
  end
end
```

В отчете команды rspec в этом случае для отложенного теста вместе с точкой ставится звездочка:

```
$ rspec spec
...*

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) Пример отложен
     # Temporarily skipped with xit
     # ./spec/example_spec.rb:8

Finished in 0.00643 seconds (files took 0.13514 seconds to load)
4 examples, 0 failures, 1 pending
```

В конце отчета описываются все отложенные тесты.

У метода describe имеется синоним: context. Более того, методы describe и context могут быть вложены друг в друга (листинг 32.12).

Листинг 32.12. Использование метода context. Файл rspec/spec/context_spec.rb

```
RSpec.describe 'Объект' do
  context 'ключ-значение' do
  end
  context 'строки' do
  end
  context 'пользователя' do
  end
end
```

Внутри каждого из контекстов можно разместить свои собственные тесты, тем самым группируя их по назначению.

Фреймворк RSpec предоставляет возможность выполнять код до и после тестов, используя колбэки before и after. В листинге 32.13 при помощи колбэка before создается инстанс-переменная @object, которая затем может быть использована внутри it-теста.

> ***Замечание***
>
> Метод, который неявно вызывается перед или после вызова другого метода, в RSpec принято называть *колбэком* (callback).

Листинг 32.13. Использование метода before. Файл rspec/spec/before_spec.rb

```
require 'ostruct'

RSpec.describe 'Объект' do
  context 'ключ-значение' do
```

```
    before(:each) do
      @object = OpenStruct.new(key: 'ключ', value: 'значение')
    end

    it 'должен содержать корректный ключ' do
      expect(@object.key).to eq 'ключ'
    end
  end
end
```

Метод `before` может принимать два параметра: `:each` и `:all`. В случае `:each` блок метода будет выполняться перед каждым тестом текущей `context/describe`-области. В случае `:all` содержимое блока будет выполнено один раз перед всеми тестами.

Объект `@object` создается при помощи класса `OpenStruct` (см. *разд. 25.3*) и содержит атрибуты: `key` и `value`.

Кроме того, можно использовать специальный метод `let`, который предоставляет «ленивую» загрузку, — объект загружается только в том случае, если к нему есть обращения, а если обращений нет, то загрузка не происходит (листинг 32.14).

Листинг 32.14. Использование метода *let*. Файл rspec/spec/let_spec.rb (предварительно)

```
require 'ostruct'

RSpec.describe 'Объект' do
  context 'ключ-значение' do
    let(:object) do
      OpenStruct.new(key: 'ключ', value: 'значение')
    end

    it 'должен содержать корректный ключ' do
      expect(object.key).to eq 'ключ'
    end
  end
end
```

Метод `let` вызывается только тогда, когда интерпретатор доходит до вызова `object.key`. Если в тестах не будет обращения к `object`, то `OpenStruct`-объект вообще не будет создан.

В листинге 32.15 создается дополнительный вложенный контекст, в котором тестируется изменение ключа `key`. Для изменения ключа удобно воспользоваться `before`-колбэком, внутри которого изменяется значение `object.key`.

Листинг 32.15. Тестирование изменения ключа. Файл rspec/spec/let_spec.rb (окончательно)

```
require 'ostruct'

RSpec.describe 'Объект' do
  context 'ключ-значение' do
    let(:object) do
      OpenStruct.new(key: 'ключ', value: 'значение')
    end

    it 'должен содержать корректный ключ' do
      expect(object.key).to eq 'ключ'
    end

    context 'с измененным значением' do
      before(:each) do
        object.key = 'новый ключ'
      end

      it 'должен содержать новый ключ' do
        expect(object.key).to eq 'новый ключ'
      end
    end
  end
end
```

В приведенном примере внутри `it`-теста происходит проверка, возвращает ли `object.key` новое значение.

32.3.3. Фреймворк *Cucumber*

Основная специализация фреймворка `Cucumber` — описание поведения приложения. Очень часто этот фреймворк используется для создания приемочных тестов. Более того, `Cucumber` позволяет писать тестовые сценарии на обычном, в том числе, русском языке.

В листинге 32.16 представлен конфигурационный файл Gemfile гема `bundler`. В нем устанавливается единственный гем: `Cucumber`.

Листинг 32.16. Установка гема `Cucumber`. Файл cucumber/Gemfile

```
source 'https://rubygems.org'
gem 'cucumber'
```

Для установки гема необходимо запустить команду `bundle` без параметров:

```
$ bundle
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
```

```
Fetching backports 3.11.4
Installing backports 3.11.4
Fetching builder 3.2.3
Installing builder 3.2.3
Using bundler 1.16.0
Fetching cucumber-tag_expressions 1.1.1
Installing cucumber-tag_expressions 1.1.1
Fetching gherkin 5.1.0
Installing gherkin 5.1.0
Fetching cucumber-core 3.2.1
Installing cucumber-core 3.2.1
Fetching cucumber-expressions 6.0.1
Installing cucumber-expressions 6.0.1
Fetching cucumber-wire 0.0.1
Installing cucumber-wire 0.0.1
Fetching diff-lcs 1.3
Installing diff-lcs 1.3
Fetching multi_json 1.13.1
Installing multi_json 1.13.1
Fetching multi_test 0.1.2
Installing multi_test 0.1.2
Fetching cucumber 3.1.2
Installing cucumber 3.1.2
Bundle complete! 1 Gemfile dependency, 12 gems now installed.
Bundled gems are installed into `./vendor/bundle`
```

Инициализация проекта выполняется командой `cucumber --init`:

```
$ cucumber --init
  create   features
  create   features/step_definitions
  create   features/support
  create   features/support/env.rb
```

Команда создает каталог features, в котором сосредотачиваются файлы-заготовки. Для создания теста в этом каталоге необходимо создать файл с расширением feature (листинг 32.17).

Листинг 32.17. Cucumber-тест. Файл cucumber/features/example.feature

```
# language: ru

Функционал: пусть класс поприветствует нас
  Для того, чтобы изучить Cucumber,
  Хорошо бы увидеть, как Cucumber работает на практике
  Напишем простейший Hello World

  Сценарий: пусть класс скажет Привет
    Допустим, дан объект класса
```

```
    Если я вызываю его метод say,
    То получаю строку 'Привет!'
```

Для того чтобы создавать файлы на русском языке, необходимо указать текущий язык при помощи директивы `language`. Далее описывается то, что подвергается тестированию, — для этого используется ключевое слово `Функционал` с его описанием. Для задания поведения функционала служит ключевое слово `Сценарий`. Предложения сценария начинаются с ключевых слов: `Если`, `Допустим` и `То`.

Запустить тест на выполнение можно при помощи утилиты `cucumber`, в качестве параметра команда принимает путь к каталогу с тестами:

```
$ cucumber features
# language: ru
Функционал: пусть класс поприветствует нас
  Для того, чтобы изучить Cucumber,
  Хорошо бы увидеть, как Cucumber работает на практике
  Напишем простейший Hello World

  Сценарий: пусть класс скажет Привет # features/example.feature:8
    Допустим, дан объект класса       # features/example.feature:9
    Если я вызываю его метод say,     # features/example.feature:10
    То получаю строку 'Привет!'       # features/example.feature:11

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.006s

You can implement step definitions for undefined steps with these snippets:

Допустим("дан объект класса") do
  pending # Write code here that turns the phrase above into ...
end

Если("я вызываю его метод say") do
  pending # Write code here that turns the phrase above into ...
end

То("получаю строку {string}") do |string|
  pending # Write code here that turns the phrase above into ...
end
```

Сценарий отработал, но так как он не запрограммирован, тесты пока не выполняются. Более того, фреймворк предлагает возможную реализацию сценария: готовые методы `Допустим`, `Если` и `То`.

Вместо строк, которые передаются методам в качестве параметров, можно использовать регулярные выражения (см. *главу 31*). Методы сценария реализуются в каталоге step_definitions. В листинге 32.18 приводится файл hello_steps.rb из этого каталога.

Листинг 32.18. Сценарий. Файл cucumber/features/step_definitions/hello_steps.rb

```
class Hello
  def say
    'Привет!'
  end
end

Допустим /^дан объект класса$/ do
  @hello = Hello.new
end

Если /^я вызываю его метод say$/ do
  @msg = @hello.say
end

То /^получаю строку '([^']*)'$/ do |str|
  @msg == str
end
```

Тестируемый класс размещен прямо в файле hello_steps.rb. На практике классы подключают при помощи методов `require` или `require_relative`.

Далее необходимо реализовать шаги нашего сценария: вызываем методы `Допустим`, `Если` и `То`. Внутри блока метода `Допустим` объявляем инстанс-переменную `@hello`, которую инициализируем объектом класса `Hello`.

В блоке метода `Если` вызывается метод `@hello.say`, результат вызова сохраняется в другую инстанс-переменную: `@msg`.

Последним шагом вызывается метод `То`, который при помощи регулярного выражения извлекает из сценария строку `'Привет!'` и передает ее внутрь блока в параметре `str`. Внутри блока полученная строка сравнивается с переменной `@msg`, в которой хранится результат вызова метода `say`.

Теперь, если запустить команду `cucumber`, можно обнаружить, что тесты успешно выполняются:

```
$ cucumber features
# language: ru
Функционал: пусть класс поприветствует нас
  Для того, чтобы изучить Cucumber,
  Хорошо бы увидеть, как Cucumber работает на практике
  Напишем простейший Hello World

  Сценарий: пусть класс скажет Привет # features/example.feature:8
    Допустим дан объект класса        #
features/step_definitions/hello_steps.rb:7
    Если я вызываю его метод say      #
features/step_definitions/hello_steps.rb:11
```

```
    То получаю строку 'Привет!'       #
features/step_definitions/hello_steps.rb:15


1 scenario (1 passed)
3 steps (3 passed)
0m0.003s
```

Таким образом, при помощи `Cucumber` был описан функционал программы и создан сценарий поведения. Причем спецификация программы получилась на русском языке и доступна для редактирования не только разработчикам, но и людям, далеким от программирования.

Задания

1. Создайте класс пользователя `User`, объект которого сможет хранить фамилию, имя, отчество пользователя, а также его электронный адрес. При помощи фреймворка `RSpec` протестируйте класс и его методы.
2. Для класса `User` из предыдущего задания создайте `Cucumber`-тесты, которые описывают поведение объекта этого класса.

Заключение

Книга завершена, однако ваше обучение на этом не заканчивается. За границами книги осталось множество неохваченных областей, которые вам придется штурмовать при помощи других книг или самостоятельно. Вот далеко не полный список технологий, языков программирования, баз данных и фреймворков, которые невозможно было детально описать на страницах книги, но которые будут полезны Ruby-разработчику:

- основы UNIX-подобных операционных систем;
- шаблоны проектирования;
- система контроля Git;
- протокол HTTP;
- протокол, серверы и клиенты SSH;
- Ruby-подобный язык Elixir на базе Erlang;
- детальное знакомство с современными возможностями SQL;
- реляционная база данных PostgreSQL;
- NoSQL, базы данных Redis, ElasticSearch, ClickHouse и Cassandra;
- язык разметки HTML;
- каскадные таблицы стилей CSS;
- язык программирования JavaScript;
- веб-сервер nginx;
- веб-фреймворки Ruby on Rails, Sinatra и Hanami;
- шаблонизаторы HAML и Slim;
- тестирование с использованием фреймворков RSpec, MiniTest и Cucumber;
- очереди на базе Redis и RabbitMQ;

- обработка изображений средствами библиотек GDLib и ImageMagic;
- технологии виртуализации: VirtualBox, Docker, Kubernetes.

Кроме того, следует помнить, что основной залог успеха в программировании — это собственно кодирование, написание кода. Только в процессе разработки реальных проектов приходит глубокое понимание технологий, причин их возникновения, преимуществ и трудностей использования в той или иной ситуации.

Удачи и успехов в программировании!

ПРИЛОЖЕНИЕ 1

Справочные таблицы

В этом приложении собраны справочные сведения о языке Ruby. Из-за последовательного изложения материала многие сведения зачастую разбросаны по всей книге. Из-за этого ко многим темам приходится возвращаться многократно. Здесь синтаксис языка сведен в табличную форму.

П1.1. Ключевые слова

В Ruby ключевые слова часто трудно отличить от методов и классов. Например, `require` и `include` — методы, а `yield` и `alias` — ключевые слова. В табл. П1.1 приводится список всех ключевых слов языка Ruby.

> ***Замечание***
>
> Список ключевых слов всегда можно уточнить в исходных кодах языка Ruby — см. файл по ссылке **https://github.com/ruby/ruby/blob/trunk/defs/keywords**.

Таблица П1.1. *Ключевые слова*

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

П1.2. Синтаксические конструкторы

Часть объектов языка Ruby могут быть созданы при помощи специальных синтаксических конструкторов (табл. П1.2).

Таблица П1.2. *Синтаксические конструкторы*

Класс	Синтаксический конструктор	Описание
`String`	`'Hello wold!'`	Строки
`Symbol`	`:white`	Символы
`Integer`	`15`	Целые числа
`Float`	`3.14159`	Числа с плавающей точкой
`Range`	`1..10`	Диапазон
`Array`	`[1, 2, 3, 4]`	Массив
`Hash`	`{hello: 'world', lang: 'ruby'}`	Хэш
`Proc`	`->(a, b) { a + b }`	Прок-объекты
`Regexp`	`//`	Регулярное выражение
`NilClass`	`nil`	Неопределенное значение
`TrueClass`	`true`	Логическая истина
`FalseClass`	`false`	Логическая ложь

Помимо синтаксических конструкторов, приведенных в табл. П1.2, Ruby предоставляет синтаксические конструкторы, которые начинаются с символа `%` (табл. П1.3). Для каждого из синтаксических конструкторов предусмотрен вариант с прописной (заглавной) буквы, в котором допускается интерполяция переменных (см. *разд. 4.2.1*).

Таблица П1.3. *Синтаксические конструкторы, начинающиеся с символа* `%`

Синтаксический конструктор	Описание
`%q`	Строка
`%s`	Символ
`%w`	Массив строк
`%i`	Массив символов
`%r`	Регулярное выражение
`%x`	Строка, полученная выполнением команды

Для формирования многострочных строк предусмотрен специальный heredoc-оператор (см. *разд. 4.2.2*). Различные типы этого оператора представлены в табл. П1.4.

Таблица П1.4. *Синтаксис heredoc-оператора*

Оператор	Описание
`<<here`	Обычный heredoc-оператор
`<<-here`	Допускаются пробельные символы перед завершающей меткой `here`
`<<~here`	Убираются любые пробельные символы перед всеми строками
`<<~'here'`	Подавляется интерполяция Ruby-выражений в строке

П1.3. Экранирование

В строках обратный слеш придает некоторым символам специальное значение (табл. П1.5).

Таблица П1.5. *Специальные символы и их значения*

Значение	Описание
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Символ табуляции
`\\`	Обратный слеш
`\"`	Двойная кавычка
`\'`	Одинарная кавычка

П1.4. Переменные и константы

В Ruby различают четыре типа переменных (табл. П1.6), детальное описание которых можно найти в *главе 5*.

Таблица П1.6. *Типы переменных*

Переменная	Описание
`local`	Локальная переменная
`$global`	Глобальная переменная
`@object`	Переменная объекта или инстанс-переменная
`@@klass`	Переменная класса

Для извлечения полного списка определенного типа переменных используются методы: `local_variables`, `global_variables`, `instance_variables` и `class_variables`.

В коде не поощряется использование глобальных переменных, тем не менее в программах часто можно встретить предопределенные глобальные переменные (табл. П1.7).

Таблица П1.7. *Предопределенные глобальные переменные*

Переменная	Описание
`$!`	Последний объект-исключение, вызванное методом `raise`
`$@`	Массив с цепочкой вызова для последнего исключения
`$&`	Строка, соответствующая последнему удачному сопоставлению регулярному выражению
`$``	Часть строки, предшествующая сопоставлению регулярному выражению
`$'`	Часть строки после сопоставления регулярному выражению
`$+`	Соответствие последним круглым скобкам
`$1`	Соответствие первым круглым скобкам (вместо цифры `1` можно использовать `2`, `3` и т. д.)
`$~`	Объект `MatchData` последнего регулярного выражения
`$/`	Разделитель строки, по умолчанию: `\n`
`$\`	Разделитель для аргументов в методе `IO#wrire`. По умолчанию принимает значение `nil`
`$,`	Разделитель для аргументов в методах `print` и `Array#join`. По умолчанию принимает значение `nil`
`$;`	Разделитель по умолчанию для метода `String#split`. По умолчанию: пробел. Синоним для `$-F`
`$.`	Номер строки последнего прочитанного файла
`$<`	Синоним для предопределенной константы `ARGF`
`$>`	Поток вывода для `print` и `printf`. По умолчанию это `$stdout`
`$_`	Последняя строка, полученная методами `gets` или `readline`
`$0`	Имя текущей программы
`$*`	Массив аргументов командной строки
`$$`	Pid-идентификатор текущего Ruby-процесса
`$?`	Статус последнего выполненного дочернего процесса
`$:`	Массив путей до Ruby-библиотек, которые используются при выполнении методов: `load`, `require` и `require_relative`. Синоним: `$LOAD_PATH` и `$-I`
`$"`	Массив модулей, загруженных методами: `require` и `require_relative`. Синоним: `$LOADED_FEATURES`

Таблица П1.7 *(окончание)*

Переменная	Описание
`$DEBUG`	Принимает значение `true`, если Ruby-интерпретатор запускается в отладочном режиме. Для запуска в этом режиме необходимо использовать параметр `-d`. Синоним: `$-d`
`$LOADED_FEATURES`	Массив модулей, загруженных методами: `require` и `require_relative`. Синоним: `$"`
`$FILENAME`	Имя текущей Ruby-программы
`$LOAD_PATH`	Массив путей до Ruby-библиотек, которые используются при выполнении методов: `load`, `require` и `require_relative`. Синоним: `$:` и `$-I`
`$stderr`	Стандартный поток для вывода ошибок
`$stdin`	Стандартный поток ввода
`$stdout`	Стандартный поток вывода
`$VERBOSE`	Подробный вывод замечаний в стандартный поток вывода. Включается передачей Ruby-интерпретатору параметров: `-w` или `-v`. Синонимы: `$-v` и `$-w`
`$-0`	Разделитель строки, по умолчанию: `\n`. Синоним для `$/`
`$-a`	Принимает значение `true`, если Ruby-интерпретатор запускается с параметром `-a`
`$-d`	Принимает значение `true`, если Ruby-интерпретатор запускается в отладочном режиме. Для запуска в этом режиме необходимо использовать параметр `-d`. Синоним для `$DEBUG`
`$-F`	Разделитель по умолчанию для метода `String#split`. Синоним для `$;`
`$-i`	Содержит список загруженных расширений при запуске Ruby-интерпретатора в однострочном режиме: `ruby -idate -e 'puts $-i'`
`$-I`	Массив путей до Ruby-библиотек, которые используются при выполнении методов: `load`, `require` и `require_relative`. Синоним: `$LOAD_PATH` и `$:`
`$-l`	Принимает значение `true`, если Ruby-интерпретатор запускается с параметром `-l`
`$-p`	Принимает значение `true`, если Ruby-интерпретатор запускается с параметром `-p`
`$-v`	Подробный вывод замечаний в стандартный поток вывода. Включается передачей Ruby-интерпретатору параметра `-v`. Синонимы: `$-w` и `$VERBOSE`
`$-w`	Подробный вывод замечаний в стандартный поток вывода. Включается передачей Ruby-интерпретатору параметра `-w`. Синонимы: `$-v` и `$VERBOSE`

В табл. П1.8 представлены предопределенные константы.

Таблица П1.8. *Предопределенные константы Ruby*

Константа	Описание
RUBY_VERSION	Версия Ruby
RUBY_RELEASE_DATE	Строка с датой релиза Ruby
RUBY_PLATFORM	Строка с идентификатором операционной системы
ARGF	Поток, читающий данные из командной строки или стандартного потока ввода
ARGV	Аргументы, переданные программе
ENV	Переменные окружения
STDOUT	Поток стандартного вывода
STDIN	Поток стандартного ввода
STDERR	Поток ошибок
DATA	Содержимое Ruby-программы после ключевого слова __END__

П1.5. Операторы

Операторы по своему назначению делятся на несколько групп: арифметические (табл. П1.9), логические (табл. П1.10), поразрядные (табл. П1.11) и операторы сравнения (табл. П1.12).

Таблица П1.9. *Арифметические операторы*

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
**	Возведение в степень

Таблица П1.10. *Логические операторы*

Оператор	Описание
x && y	Логическое И, возвращает true, если оба операнда: x и y — истинны, в противном случае возвращается false
x and y	Логическое И, отличающееся от оператора && меньшим приоритетом

Таблица П1.10 *(окончание)*

Оператор	Описание
`x || y`	Логическое ИЛИ, возвращает `true`, если хотя бы один из операндов: `x` или `y` — истинен. Если оба операнда ложны, оператор возвращает `false`
`x or y`	Логическое ИЛИ, отличающееся от оператора `||` меньшим приоритетом
`! x`	Возвращает либо `true`, если `x` ложен, либо `false`, если `x` истинен

Таблица П1.11. *Поразрядные операторы*

Оператор	Описание
`&`	Поразрядное пересечение: И (AND)
`|`	Поразрядное объединение: ИЛИ (OR)
`^`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)
`~`	Поразрядное отрицание (NOT)
`<<`	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда
`>>`	Сдвиг вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда

Таблица П1.12. *Операторы сравнения*

Оператор	Описание
`>`	Оператор «больше», возвращает `true`, если левый операнд больше правого
`<`	Оператор «меньше», возвращает `true`, если левый операнд меньше правого
`>=`	Оператор «больше равно», возвращает `true`, если левый операнд больше или равен правому операнду
`<=`	Оператор «меньше равно», возвращает `true`, если левый операнд меньше или равен правому операнду
`==`	Оператор равенства, возвращает `true`, если сравниваемые операнды равны
`!=`	Оператор неравенства, возвращает `true`, если сравниваемые операнды не равны
`<=>`	Возвращает `-1`, если левый операнд меньше правого, `0` — в случае, если операнды равны, и `1`, если левый операнд больше правого
`=~`	Возвращает `true`, если операнд соответствует регулярному выражению
`!~`	Возвращает `true`, если операнд не соответствует регулярному выражению
`===`	Оператор равенства, предназначенный для перегрузки в классах, используется явно и при сравнении в конструкции `case`

При сравнении объектов друг с другом следует помнить, что объекты считаются равными друг другу, если это один и тот же объект. По умолчанию согласно этому правилу сравнивают операторы == и equal?. В табл. П1.13 собраны все возможные способы сравнения объектов.

Таблица П1.13. *Способы сравнения объектов*

Оператор	Описание
==	Сравнение объектов на равенство
===	Оператор равенства, предназначенный для перегрузки в классах, используется явно и при сравнении в конструкции case
equal?	Метод equal?, в отличие от оператора ==, не принято перегружать, он всегда должен позволять сравнивать объекты классическим способом
eql?	Используется при сравнении ключей хэша

Операторы имеют приоритет (табл. П1.14). В таблице операторы расположены в соответствии с уменьшением их приоритета сверху вниз. Операторы, расположенные в ячейке таблицы на одной строке, имеют одинаковый приоритет, и выполняется в первую очередь тот из операторов, который встречается в выражении первым.

Таблица П1.14. *Приоритет операторов*

Оператор	Описание
! ~ +	Логическое отрицание !, поразрядное отрицание ~, унарный плюс +. Например: +10
**	Возведение в степень
-	Унарный минус -. Например: -10
* / %	Умножение *, деление /, взятие остатка %
+ -	Арифметические плюс + и минус -
>> <<	Поразрядные сдвиги вправо >> и влево <<
&	Поразрядное И
^ |	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ и поразрядное ИЛИ
<= < > >=	Операторы сравнения
<=> == === != =~ !~	Операторы сравнения
&&	Логическое И
||	Логическое ИЛИ
.. ...	Операторы диапазона
z ? y : z	Тернарный оператор
= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	Операторы присваивания

Таблица П1.14 *(окончание)*

Оператор	Описание
`not`	Логическое отрицание
`or and`	Логические ИЛИ и И

П1.6. Конструкции ветвления и циклы

Конструкции ветвления представлены в табл. П1.15, а в табл. П.16 — циклы.

Таблица П1.15. *Конструкции ветвления*

Конструкция	Описание
`if`	Выполняет блок, если аргумент — истинное выражение
`unless`	Выполняет блок, если аргумент — ложное выражение
`case`	Выполняет множественное сравнение величины с различными вариантами, для сравнения используется оператор ===

Таблица П1.16. *Циклы*

Циклы	Описание
`while`	Выполняет блок цикла до тех пор, пока условие истинно
`until`	Выполняет блок цикла до тех пор, пока условие ложно
`for ... in ...`	Выполняет цикл до тех, пор, пока не исчерпаются элементы в коллекции

Для управления циклами Ruby предоставляет ряд ключевых слов (табл. П1.17).

Таблица П1.17. *Управление циклами*

Ключевое слово	Описание
`break`	Досрочный выход из цикла
`next`	Досрочное прекращение текущей итерации
`redo`	Повторение текущей итерации

П1.7. Итераторы

В разных классах могут быть реализованы различные итераторы. Однако наибольшей популярностью пользуются итераторы, реализованные в модуле `Enumerable` (табл. П1.18).

Таблица П1.18. *Итераторы модуля* `Enumerable`

Итератор	Синоним
`each`	
`each_with_index`	
`map`	`collect`
`select`	`find_all`
`reject`	`delete_if`
`reduce`	`inject`
`each_with_object`	

ПРИЛОЖЕНИЕ 2

Содержимое электронного архива

Электронный архив, сопровождающий книгу, содержит все приведенные в ее листингах исходные коды (табл. П2.1).

Сам электронный архив к книге выложен на FTP-сервер издательства «БХВ-Петербург» по адресу: **ftp://ftp.bhv.ru/9785977540605.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Исходные коды к книге также размещены на GitHub-аккаунте по адресу: **https://github.com/igorsimdyanov/ruby**.

Таблица П2.1. *Содержимое электронного архива*

Каталог	Описание
start	Примеры *главы 2. Быстрый старт*
utils	Примеры *главы 3. Утилиты и гемы*
ruby_classes	Примеры *главы 4. Предопределенные классы*
variables	Примеры *главы 5. Переменные*
constants	Примеры *главы 6. Константы*
operators	Примеры *главы 7. Операторы*
conditions	Примеры *главы 8. Ветвление*
methods	Примеры *главы 9. Глобальные методы*
cycles	Примеры *главы 10. Циклы*
iterators	Примеры *главы 11. Итераторы*
blocks	Примеры *главы 12. Блоки*
classes	Примеры *главы 13. Классы*
class_methods	Примеры *главы 14. Методы в классах*
overload	Примеры *главы 15. Преобразование объектов*
self	Примеры *главы 16. Ключевое слово* `self`
inheritance	Примеры *главы 17. Наследование*

Таблица П2.1 *(окончание)*

Каталог	Описание
scope	Примеры *главы 18. Области видимости*
namespaces	Примеры *главы 19. Модули*
mixins	Примеры *главы 20. Подмешивание модулей*
modules	Примеры *главы 21. Стандартные модули*
objects	Примеры *главы 22. Свойства объектов*
arrays	Примеры *главы 23. Массивы*
hashes	Примеры *главы 24. Хэши*
collections	Примеры *главы 25. Классы коллекций*
exceptions	Примеры *главы 26. Исключения*
files	Примеры *главы 27. Файлы*
files_attributes	Примеры *главы 28. Права доступа и атрибуты файлов*
catalogs	Примеры *главы 29. Каталоги*
regexp	Примеры *главы 30. Регулярные выражения*
rack	Примеры *главы 31. Веб-программирование*
tests	Примеры *главы 32. Автоматическое тестирование*

Предметный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

А

Б

В

Г

Д

Ж

З

И

К

Л

M

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю