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*

Подписано в печать 31.07.19.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 52,89.
Тираж 1000 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-4060-5

© ООО "БХВ", 2020
© Оформление. ООО "БХВ-Петербург", 2020

Оглавление

Предисловие ... **13**
Цель книги ... 13
Как создавалась книга... ... 13
Терминология ... 14
Исходные коды ... 14
Задания ... 15
Типографские соглашения ... 15
Благодарности ... 16

Глава 1. Введение в язык Ruby ... **19**
1.1. Философия Ruby ... 19
1.2. Реализации Ruby ... 21
1.3. Версии ... 23
1.4. Установка Ruby ... 24
1.4.1. Установка Ruby в Windows ... 24
1.4.2. Установка Ruby в Linux (Ubuntu) ... 26
1.4.2.1. Менеджер версий RVM ... 26
1.4.2.2. Менеджер версий rbenv ... 28
1.4.3. Установка Ruby в macOS ... 29
1.5. Запуск программы на выполнение ... 30
Задания ... 31

Глава 2. Быстрый старт ... **33**
2.1. Соглашения Ruby ... 33
2.2. Комментарии ... 34
2.3. Элементы языка ... 35
2.3.1. Ключевые слова ... 36
2.3.2. Переменные ... 37
2.3.3. Константы ... 37
2.3.4. Объекты ... 38
2.3.5. Классы и модули ... 39
2.3.6. Методы ... 40
2.3.7. Операторы ... 40

2.4. Вывод в стандартный поток 41
2.4.1. Вывод при помощи методов *puts* и *p* 41
2.4.2. Экранирование 42
2.5. Как пользоваться документацией? 43
2.5.1. Справочные методы объектов 44
2.5.2. Консольная справка 45
2.5.3. Online-документация 46
Задания 46

Глава 3. Утилиты и гемы 47
3.1. Утилиты 47
3.2. Интерактивный Ruby 48
3.3. Шаблонизатор *erb* 50
3.4. Утилита *rake* 50
3.5. Утилита *rdoc* 53
3.6. Гемы 54
3.6.1. Отладка программ при помощи гема *pry* 56
3.6.2. Контроль стиля кода при помощи гема *rubocop* 57
3.6.3. Управление гемами при помощи *bundler* 58
Задания 63

Глава 4. Предопределенные классы 65
4.1. Синтаксические конструкторы 65
4.2. Строки. Класс *String* 67
4.2.1. Синтаксические конструкторы *%q* и *%Q* 67
4.2.2. *Heredoc*-оператор 68
4.2.3. Выполнение команд операционной системы 69
4.2.4. Устройство строки 70
4.2.5. Обработка подстрок 71
4.3. Символы. Класс *Symbol* 73
4.4. Целые числа. Класс *Integer* 74
4.5. Вещественные числа. Класс *Float* 76
4.6. Диапазоны. Класс *Range* 79
4.7. Массивы. Класс *Array* 81
4.7.1. Создание массива 81
4.7.2. Операции с массивами 81
4.7.3. Синтаксические конструкторы *%w* и *%i* 82
4.8. Хэши. Класс *Hash* 83
4.9. Логические объекты *true* и *false* 84
4.10. Объект *nil* 85
Задания 85

Глава 5. Переменные 87
5.1. Типы переменных 87
5.1.1. Локальные переменные 87
5.1.2. Глобальные переменные 89
5.1.2.1. Предопределенная переменная *$LOAD_PATH* 91
5.1.2.2. Предопределенная переменная *$stdout* 91
5.1.2.3. Предопределенная переменная *$PROGRAM_NAME* 92

5.1.3. Инстанс-переменные 93
5.1.4. Переменные класса 96
5.2. Присваивание 97
5.3. Клонирование 99
Задания 101

Глава 6. Константы 103
6.1. Создание и определение констант 103
6.2. Предопределенные константы 104
6.3. Ключевые слова *__LINE__* и *__FILE__* 107
6.4. Метод *require* 107
6.5. Метод *require_relative* 109
6.6. Подключение стандартных классов 110
6.7. Подключение гемов 111
Задания 113

Глава 7. Операторы 115
7.1. Операторы — это методы 115
7.2. Арифметические операторы 116
7.3. Присваивание 117
7.3.1. Сокращенная форма арифметических операторов 117
7.3.2. Параллельное присваивание 118
7.3.3. Круглые скобки в параллельном присваивании 119
7.3.4. Оператор * 120
7.4. Операторы строк 122
7.4.1. Умножение строки на число 122
7.4.2. Сложение строк 123
7.4.3. Форматирование строк 124
7.5. Операторы сравнения 128
7.5.1. Особенности сравнения объектов 129
7.5.2. Сравнение с нулем 131
7.5.3. Особенности сравнения вещественных чисел 132
7.5.4. Особенности сравнения строк 132
7.6. Поразрядные операторы 134
7.7. Оператор безопасного вызова 138
7.8. Ключевое слово *defined?* 139
7.9. Приоритет операторов 140
Задания 141

Глава 8. Ветвление 143
8.1. Ключевое слово *if* 143
8.1.1. Ключевые слова *else* и *elsif* 145
8.1.2. Ключевое слово *then* 146
8.1.3. *if*-модификатор 147
8.1.4. Присваивание *if*-результата переменной 148
8.1.5. Присваивание в условии *if*-оператора 149
8.2. Логические операторы 150
8.2.1. Логическое И. Оператор *&&* 151
8.2.2. Логическое ИЛИ. Оператор || 151
8.2.3. Логическое отрицание 154

8.3. Ключевое слово *unless* 155
8.4. Условный оператор 156
8.5. Ключевое слово *case* 156
8.6. Советы 159
Задания 161

Глава 9. Глобальные методы 163
9.1. Создание метода 163
9.2. Параметры и аргументы 164
    9.2.1. Значения по умолчанию 165
    9.2.2. Неограниченное количество параметров 165
    9.2.3. Позиционные параметры 167
    9.2.4. Хэши в качестве параметров 167
9.3. Возвращаемое значение 168
9.4. Получатель метода 170
9.5. Псевдонимы методов 172
9.6. Удаление метода 172
9.7. Рекурсивные методы 172
9.8. Предопределенные методы 175
    9.8.1. Чтение входного потока 175
    9.8.2. Остановка программы 177
    9.8.3. Методы-конструкторы 179
9.9. Логические методы 180
9.10. bang-методы 181
Задания 182

Глава 10. Циклы 183
10.1. Цикл *while* 183
10.2. Вложенные циклы 188
10.3. Досрочное прекращение циклов 189
10.4. Цикл *until* 191
10.5. Цикл *for* 192
Задания 193

Глава 11. Итераторы 195
11.1. Итераторы и блоки 195
11.2. Обход итераторами массивов и хэшей 196
11.3. Итератор *times* 197
11.4. Итераторы *upto* и *downto* 198
11.5. Итераторы коллекций 198
    11.5.1. Итератор *each* 199
    11.5.2. Итератор *each_with_index* 200
    11.5.3. Итератор *map* 200
    11.5.4. Итераторы *select* и *reject* 201
    11.5.5. Итератор *reduce* 203
    11.5.6. Итератор *each_with_object* 204
11.6. Итератор *tap* 205
11.7. Сокращенная форма итераторов 207

11.8. Досрочное прекращение итерации 207
11.9. Класс *Enumerator* 209
Задания 210

Глава 12. Блоки 211
12.1. Блоки в собственных методах 211
12.2. Передача значений в блок 213
12.3. Метод *block_given?* 215
12.4. Возврат значений из блока 215
12.5. Итератор *yield_self* 216
12.6. Передача блока через параметр 217
12.7. Различие *{ ... }* и *do ... end* 219
12.8. Блоки в рекурсивных методах 221
12.9. Класс *Proc* 225
12.10. Методы *proc* и *lambda* 226
12.11. Различия *proc* и *lambda* 227
Задания 230

Глава 13. Классы 231
13.1. Создание класса 231
13.2. Класс — это объект 232
13.3. Как проектировать классы? 234
13.4. Переопределение методов 234
13.5. Открытие класса 235
13.6. Тело класса и его свойства 237
13.7. Вложенные классы 238
13.8. Константы 240
13.9. Переменные класса 241
Задания 243

Глава 14. Методы в классах 245
14.1. Сохранение состояния в объекте 245
14.2. Установка начального состояния объекта 246
14.2.1. Метод *initialize* 247
14.2.2. Параметры метода *new* 250
14.2.3. Блоки в методе *new* 254
14.2.4. Метод *initialize* и переменные класса 255
14.2.5. Как устроен метод *new*? 255
14.3. Специальные методы присваивания 256
14.3.1. Методы со знаком равенства (=) в конце имени 257
14.3.2. Аксессоры 259
14.4. Синглетон-методы 261
14.5. Методы класса 264
14.6. Обработка несуществующих методов 266
14.6.1. Создание метода *define_method* 267
14.6.2. Перехват вызовов несуществующих методов 268
14.7. Метод *send* 270
Задания 275

Глава 15. Преобразование объектов ... 277
15.1. Сложение строк и чисел ... 277
15.2. Методы преобразования объектов ... 278
15.3. Сложение объектов ... 282
15.4. Сложение объекта и числа ... 283
15.5. Сложение объекта и строки ... 286
15.6. Сложение объекта и массива ... 289
15.7. Перегрузка *[]* и *[]=* ... 291
15.8. Перегрузка унарных операторов +, – и *!* ... 294
15.9. Какие операторы можно перегружать? ... 295
15.10. DuckType-типизация ... 297
Задания ... 300

Глава 16. Ключевое слово *self* ... 301
16.1. Ссылки на текущий объект ... 301
16.2. Значения *self* в разных контекстах ... 304
16.3. Приемы использования *self* ... 305
16.3.1. Методы класса ... 305
16.3.2. Цепочка обязанностей ... 308
16.3.3. Перегрузка операторов ... 309
16.3.4. Инициализация объекта блоком ... 310
16.3.5. Открытие класса ... 310
Задания ... 311

Глава 17. Наследование ... 313
17.1. Наследование ... 313
17.2. Логические операторы ... 316
17.3. Динамический базовый класс ... 317
17.4. Наследование констант ... 318
17.5. Иерархия стандартных классов ... 319
17.6. Переопределение методов ... 320
17.7. Удаление методов ... 322
17.8. Поиск метода ... 326
Задания ... 328

Глава 18. Области видимости ... 331
18.1. Концепция видимости ... 331
18.2. Открытые методы ... 332
18.3. Закрытые методы ... 333
18.4. Защищенные методы ... 335
18.5. Закрытый конструктор ... 337
18.6. Паттерн «Одиночка» (*Singleton*) ... 338
18.7. Вызов закрытых методов ... 341
18.8. Информационные методы ... 342
18.9. Области видимости при наследовании ... 345
18.10. Области видимости методов класса ... 346
Задания ... 348

Глава 19. Модули ... 349
19.1. Создание модуля ... 349
19.2. Оператор разрешения области видимости ... 350
19.3. Пространство имен ... 351
19.4. Вложенные классы и модули ... 352
19.5. Доступ к глобальным классам и модулям ... 357
Задания ... 361

Глава 20. Подмешивание модулей ... 363
20.1. Класс *Module* ... 363
20.2. Подмешивание модулей в класс ... 365
20.3. Подмешивание модулей в объект ... 369
20.4. Синглетон-методы модуля ... 373
20.5. Области видимости ... 376
20.6. Стандартный модуль *Kernel* ... 379
20.7. Поиск методов в модулях ... 382
20.8. Метод *prepend* ... 386
20.9. Методы обратного вызова ... 387
20.10. Уточнения ... 393
20.11. Псевдонимы методов ... 395
Задания ... 397

Глава 21. Стандартные модули ... 399
21.1. Модуль *Math* ... 399
21.2. Модуль *Singleton* ... 402
21.3. Модуль *Comparable* ... 404
21.4. Модуль *Enumerable* ... 406
21.5. Модуль *Forwardable* ... 408
21.6. Маршаллизация ... 413
21.7. JSON-формат ... 416
21.8. YAML-формат ... 418
Задания ... 421

Глава 22. Свойства объектов ... 423
22.1. Общие методы ... 423
22.2. Неизменяемые объекты ... 424
22.3. Заморозка объектов ... 425
22.4. Небезопасные объекты ... 426
Задания ... 428

Глава 23. Массивы ... 429
23.1. Модуль *Enumerable* ... 429
23.2. Заполнение массива ... 430
23.3. Извлечение элементов ... 434
23.4. Поиск индексов элементов ... 439
23.5. Случайный элемент массива ... 439
23.6. Удаление элементов ... 440
23.7. Замена элементов ... 443
23.8. Информация о массиве ... 445
23.9. Преобразование массива ... 446

23.10. Арифметические операции с массивами 447
23.11. Логические методы....448
23.12. Вложенные массивы....451
23.13. Итераторы 453
23.14. Сортировка массивов 455
Задания 456

Глава 24. Хэши....457
24.1. Создание хэша....457
24.2. Заполнение хэша....458
24.3. Извлечение элементов....459
24.4. Поиск ключа....461
24.5. Обращение к несуществующему ключу 461
24.6. Удаление элементов хэша....463
24.7. Информация о хэшах....465
24.8. Хэши как аргументы методов....467
24.9. Объединение хэшей....468
24.10. Преобразование хэшей....468
24.11. Сравнение ключей 471
24.12. Преобразование ключей хэша 474
Задания 477

Глава 25. Классы коллекций 479
25.1. Множество *Set*....479
25.2. Класс *Struct*....485
25.3. Класс *OpenStruct* 489
Задания 490

Глава 26. Исключения....493
26.1. Генерация и перехват исключений....493
26.2. Исключения — это объекты 496
26.3. Стандартные ошибки....497
26.4. Создание собственных исключений....498
26.5. Перехват исключений....500
26.6. Многократная попытка выполнить код 501
26.7. Перехват исключений: почти всегда плохо....502
26.8. Блок *ensure* 502
26.9. Блок *else*....503
26.10. Перехват исключений в блоке 504
Задания 504

Глава 27. Файлы....505
27.1. Класс *IO*....505
27.2. Создание файла....506
27.3. Режимы открытия файла....508
27.4. Закрытие файла....510
27.5. Чтение содержимого файла 512
27.6. Построчное чтение файла 512
27.7. Запись в файл 515

27.8. Произвольный доступ к файлу 517
27.9. Пути к файлам 520
27.10. Манипуляция файлами 522
Задания 523

Глава 28. Права доступа и атрибуты файлов 525
28.1. Типы файлов 525
28.2. Определение типа файла 529
28.3. Время последнего доступа к файлу 529
28.4. Права доступа в UNIX-подобной системе 531
Задания 534

Глава 29. Каталоги 535
29.1. Текущий каталог 535
29.2. Создание каталога 536
29.3. Чтение каталога 537
29.4. Фильтрация содержимого каталога 538
29.5. Рекурсивный обход каталога 539
29.6. Удаление каталога 542
Задания 542

Глава 30. Регулярные выражения 543
30.1. Как изучать регулярные выражения? 543
30.2. Синтаксический конструктор 544
30.3. Оператор =~ 545
30.4. Методы поиска 546
  30.4.1. Метод *match* 546
  30.4.2. Метод *match?* 547
  30.4.3. Метод *scan* 548
30.5. Синтаксис регулярных выражений 548
  30.5.1. Метасимволы 548
  30.5.2. Экранирование 553
  30.5.3. Квантификаторы 554
  30.5.4. Опережающие и ретроспективные проверки 556
30.6. Модификаторы 557
30.7. Где использовать регулярные выражения? 560
30.8. Примеры 562
Задания 565

Глава 31. Веб-программирование 567
31.1. Протокол HTTP 567
  31.1.1. HTTP-заголовки 570
  31.1.2. HTTP-коды ответа 570
31.2. Веб-серверы 571
31.3. Гем *Rack* 572
  31.3.1. Установка гема *Rack* 574
  31.3.2. Простейшее *Rack*-приложение 574
  31.3.3. Управление утилитой *rackup* 578
  31.3.4. Обработка несуществующих страниц 579

31.3.5. Размер HTTP-содержимого 580
31.3.6. *Proc*-объект в качестве *Rack*-приложения 582
31.3.7. Промежуточные слои (middleware) 582
31.3.8. Роутинг 583
31.3.9. Обработка статических файлов 584
31.4. Ruby on Rails 587
31.4.1. Введение в Ruby on Rails 587
31.4.2. Установка Ruby on Rails 588
31.4.3. Паттерн MVC 591
31.4.4. Структура приложения Ruby on Rails 594
31.4.4.1. Каталог app 595
31.4.4.2. Окружения 595
31.4.4.3. Каталог config 596
31.4.4.4. Каталог db 596
31.4.4.5. Каталог lib 596
31.4.4.6. Каталог public 597
31.4.4.7. Каталог test 597
31.4.5. *Rake*-задачи 597
31.4.6. Генераторы 599
31.4.7. Стартовая страница 601
31.4.8. Представления 604
Задания 606

Глава 32. Автоматическое тестирование 607
32.1. Типы тестирования 607
32.2. Преимущества и недостатки тестирования 608
32.3. Фреймворки для тестирования 609
32.3.1. Фреймворк *MiniTest* 609
32.3.2. Фреймворк *RSpec* 613
32.3.3. Фреймворк *Cucumber* 620
Задания 624

Заключение 625

Приложение 1. Справочные таблицы 627
П1.1. Ключевые слова 627
П1.2. Синтаксические конструкторы 628
П1.3. Экранирование 629
П1.4. Переменные и константы 629
П1.5. Операторы 632
П1.6. Конструкции ветвления и циклы 635
П1.7. Итераторы 635

Приложение 2. Содержимое электронного архива 637

Предметный указатель 639

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).

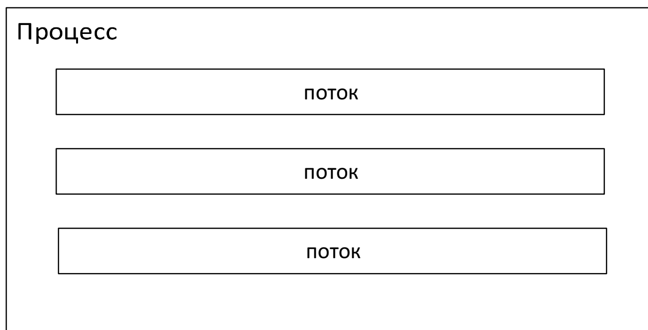


Рис. 1.1. Процессы могут содержать несколько потоков

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).

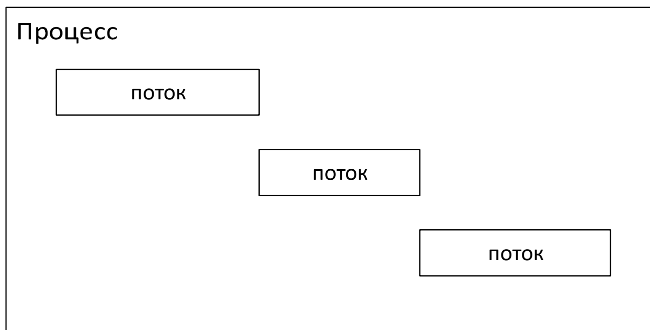


Рис. 1.2. В случае GIL в каждый момент времени может обновляться лишь один поток

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.

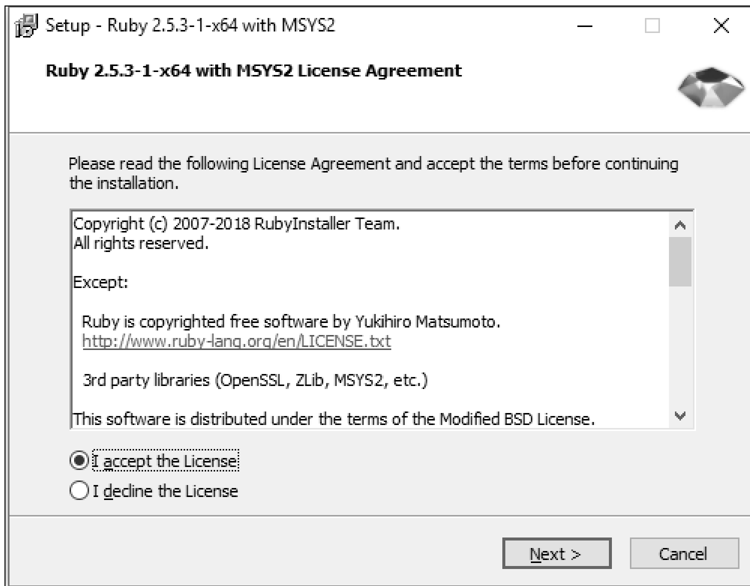


Рис. 1.4. Окно мастера установки RubyInstaller

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>.

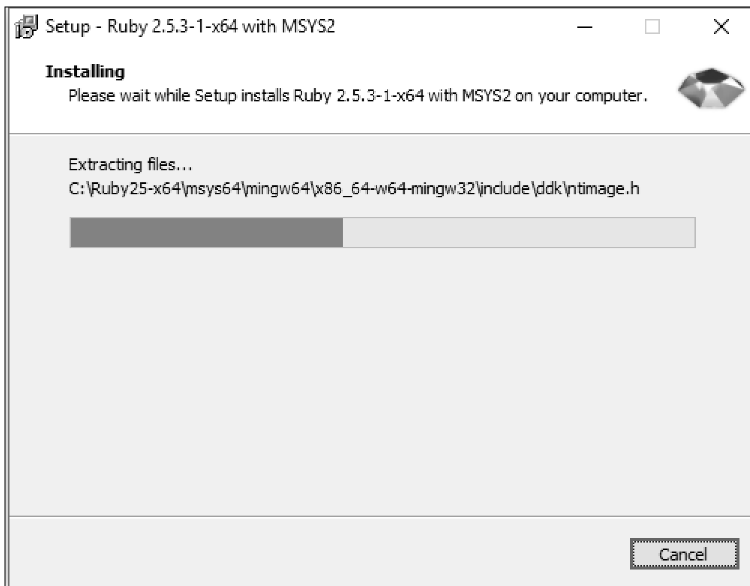


Рис. 1.5. Установка Ruby в Windows

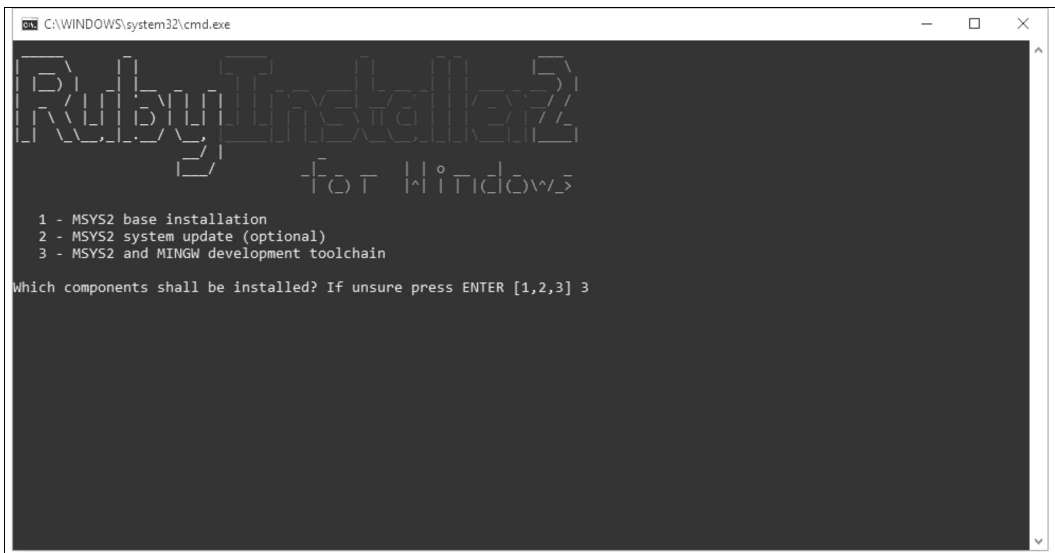


Рис. 1.6. Установка пакета MSYS2

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

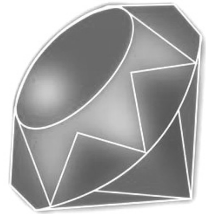
```
#!/usr/bin/env ruby
puts 'Hello, world!'
```

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.

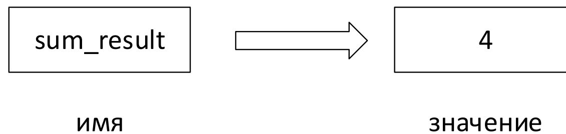


Рис. 2.1. Переменная — это значение в памяти, на которое можно сослаться при помощи имени переменной

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

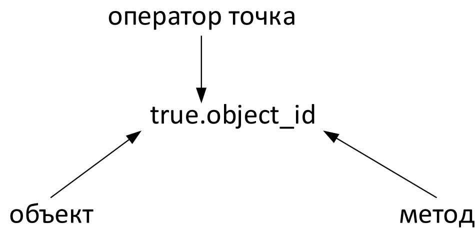


Рис. 2.2. Слева от точки всегда располагается объект, справа — метод

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).